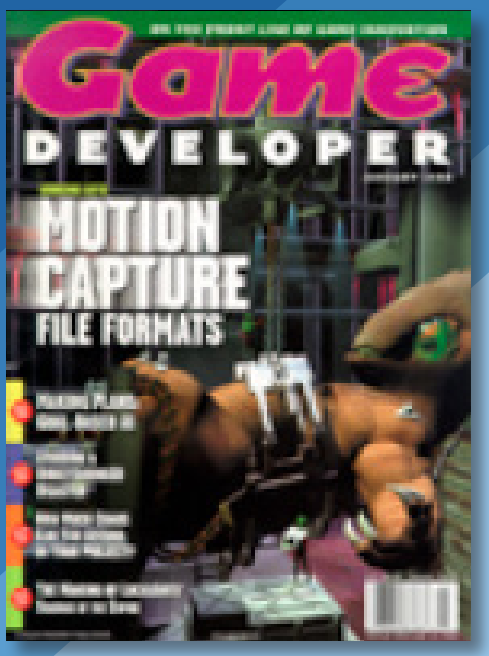
gd

GAME DEVELOPER MAGAZINE

JANUARY 1998



UBM Tech

What's Your .plan?

In Hollywood, reading an interview is the most common way to find out about what's going on in a star's life. Magazines such as *People, Rolling Stone, Spin*, and *Interview* are successful because the public wants to know what's going on with the people behind the movies and music. There's a natural curiosity about what projects they're currently working on and what's going on in their personal lives. With the exception of the rare autobiography, however, it's rare to get first-hand information from well-known stars.

That's not true in the game development industry. Game developers expound their opinions about life, the universe, and everything via .plan files, and outside of the academic world (where .plan and .project files were probably first used to keep colleagues across the country updated on research projects), this candor is fairly isolated to our industry. Some may dismiss fingering .plan files as an outdated mode of communication now that web pages are ubiquitous, but in terms of simplicity and beauty, there's nothing like pure vanilla text to get your message across.

To help disseminate the contents of the various .plan files around the industry, a number of web sites have been launched in the past two years, which effectively market .plans to the masses. A quick scan of the Stomped Finger Tracker at redwood.stomped.com reveals constantly updated .plan files from id, Rogue Entertainment, Ritual Entertainment, Raven Software, 3DRealms/Apogee, ION Storm, Quantum Axcess, and more. Over 100 developers have .plan files listed on the site, and about a quarter of those are updated every week — quite a bit of information.

At a business level, there are so many reasons to author a .plan that it's hard to know where to begin. A good .plan brings developers closer to customers, letting consumers tap into the thoughts and feelings of the people behind the games. .plans build excitement for upcoming games and connect consumers with the creators of shipping games. They bring game players closer to members of the development team, and they build company identity, loyalty, and developer name recognition. Undoubtedly, all of these factors translate in some way to increased sales.

At a personal level, authoring a .plan can raise your personal stock — if it's written consistently and intelligently. And building name recognition within the industry and with customers should be a high priority for anyone who's serious about making a name for themselves.

A word of caution, however. Remember that what the .plan giveth, the .plan can taketh away. Those who forget to engage their brains before putting mouths in action often learn tough lessons about making derogatory public statements on the Net. Off-the-cuff comments have been misconstrued, innocuous statements have been turned against authors, and 3AM rants have proven to be public relations messes the following day. As with anything you run up the flagpole on Usenet and the Web, adopting a harsh tone or making inaccurate statements can get authors and their companies in hot water. Companies whose developers author .plans should adopt guidelines that spell out which topics are acceptable to write about and which are not. This may sound authoritarian and somewhat counter to the open nature of .plan authorship, but it also prevents someone from exercising poor judgement in a .plan.

My point is this: don't overlook the obvious when you're trying to get some attention in today's crowded marketplace. Web sites are great customer service tools and online product brochures, but all too often they lack soul. If you're management, encourage .plan authorship within your development teams' ranks. If you're creating a game and your company doesn't already support the notion of .plans, approach management with the idea. Doesn't it make sense for you and your company to do everything within your power to generate consumer interest and loyalty, especially when all that's required is a little time every week and some space on your server? ■

Game Developer
ON THE FRONT LINE OF GAME INNOVATION

EDITOR IN CHIEF Alex Dunne
adunne@compuserve.com
MANAGING EDITOR Tor Berg
tdberg@sirius.com
EDITORIAL ASSISTANT Wesley Hall
whall@mfi.com
EDITOR-AT-LARGE Chris Hecker
checker@bix.com
CONTRIBUTING EDITORS Brian Hook
bwh@wksoftware.com
Josh White
josh@vectorg.com
ART DIRECTOR Azriel Hayes
ahayes@mfi.com
ADVISORY BOARD Hal Barwood
Noah Falstein
Susan Lee-Merrow
Mark Miller

COVER IMAGE Darwin 3D

PUBLISHER KoAnn Vikören
ASSOCIATE PUBLISHER Cynthia A. Blair
cblair@mfi.com
WESTERN REGIONAL SALES MANAGER Tony Andrade
(415) 905-2156
tandrade@mfi.com

MARKETING MANAGER Susan McDonald
AD. PRODUCTION COORDINATOR Dave Perrotti
DIRECTOR OF PRODUCTION Andrew A. Mickus
VICE PRESIDENT/CIRCULATION Jerry M. Okabe
GROUP CIRCULATION DIRECTOR Mike Poplardo
CIRCULATION MANAGER Stephanie Blake
CIRCULATION ASSISTANT Kausha Jackson-Crain
DIRECT MAIL MANAGER Claudia Curcio
NEWSSTAND MANAGER Eric Alekman
REPRINTS Stella Valdez
(916) 983-6971

Miller Freeman
A United News & Media publication

CEO - MILLER FREEMAN GLOBAL Tony Tillin
CHAIRMAN - MILLER FREEMAN INC. Marshall W. Freeman
PRESIDENT Donald A. Pazour
SENIOR VICE PRESIDENT/CFO Warren "Andy" Ambrose
SENIOR VICE PRESIDENTS H. Ted Bahr,
Darrell Denny,
David Nussbaum,
Galen A. Poss,
Wini D. Ragus,
Regina Starr Ridley
VICE PRESIDENT/PRODUCTION Andrew A. Mickus
VICE PRESIDENT/CIRCULATION Jerry M. Okabe
SENIOR VICE PRESIDENT/ SYSTEMS AND SOFTWARE DIVISION Regina Starr Ridley

www.gdmag.com

What Dave Said

If we are to believe Dave Thielen ("Goodbye For Now," Soapbox, October 1997), then all of us in the game industry are uneducated sloths who would do better to step aside and let the "professional" programmers do the job! If that is the way Mr. Thielen wants it, then scratch ULTIMA: It was developed by a young Richard Garriott. And scratch many other successful games while you're at it.

I have seen many games programmed by "professional" programmers. These games all have "really slick code" — and that's about it. No heart, no soul, no passion. The game industry may change, but it will never completely stamp out the Leonardos and Michaelangelos of the game industry. These are the hearty souls that other industries wish they had. They are the pioneers with the raw talent and stamina to overcome the persecution perpetrated by the arrogance so often seen in the attitudes of those who deem themselves "professional." Many of the games that people love today were developed on an extremely tight budget. How many programmers working in other industries would go the extra yard to get their product out even though they hadn't been paid for a month?

Concerning game publishers' willingness to decide for themselves whether or not a product is worthy of publishing: This problem isn't isolated to the game industry. Publishers in general have this problem. They have many products thrust at them, and when a product doesn't stand out from the crowd, it gets overlooked. Rejection is part of the game; get used to it.

Concerning unqualified programmers: Certainly there are some who are eccentric and maybe a few who are not "right" for the job. But to brand most of the industry as "unqualified" shows a lack of reasoning.

Concerning self-taught programmers: Every developer can benefit from experience working with a senior developer. How can anyone invalidate an individual's desire to learn whether it is through formal classroom or independent studies? Mr. Thielen's concept is both unreasonable and contradictory to many of the educational programs offered through mainstream technology companies. In fact, those interested enough to spend their own time learning a subject are usually more capable than those force-fed in the classroom. If the heart, soul, and passion driving the individual are missing, no amount of training or experience that the individual has will help finish the project.

Larry Dolyniuk
via E-mail

What Nancie Said

I feel compelled to write concerning Nancie S. Martin's Soapbox ("Take the Y Out of Computer Games," September 1997"). My first reaction was insult and anger, which has quieted to mere hurt feelings. Imagine her article being written by a Nathaniel S. Martin… imagine the sneering tone and condescending attitude of the article being applied to females, rather than males. Instead of this one, lonely missive in protest, you would be flooded with e-mail, demanding blood payment.

The nature of video games, to date, has been largely determined by the audience, the nature of those writing the games, and the limitations of the hardware/software. Conflict, the heart of storytelling (which is what video games are about), is difficult to program, except through violent, physical activity. Consider the complexity of the plot in an action movie vs. the plot in a love story or contemplative drama.

The video game industry was breech-born, in the spare time of real programmers doing "real" programming. It's hardly surprising that video games appealed to those writing them. If Ms. Martin wishes for there to be video games that appeal to her tastes, let her write them. One of my all-time favorite games is MYST. DOOM, and all of its spin-offs, literally leave me ill. DIABLO bores me. Granted, I do love the CRUSADER games, but consider the plot they have and the lengths to which the authors went to draw the player into that plot. Most of my all-time favorite games involve exploration rather than physical action. If her suppositions concerning what does and does not interest men, and the differences between what men and women like are valid, how can she explain my taste, as well as my friends' taste, in video games? Women's tastes in video games are, by and large, different from men's. Not better, not worse, merely different. However, the area of overlap is so large, I strongly doubt that it's necessary to specifically target the female audience.

As the hardware, software, and tools improve, so will the overall quality and complexity of games of all genres. For that small bit of software that specifically targets women, there will be plenty of programmers and designers, probably female, to create it. For the rest of it, I think the developers should concentrate on telling good stories.

Jim Williams
via E-mail

Wal-Mart and the RSAC

As one of the members of the committee who created the RSAC ratings system, I can tell you that one of our committee members met with Wal-Mart and received feedback from their management before moving forward with the system. We knew that publishers would not support a system that couldn't get them into Wal-Mart. I am therefore alarmed by your September editorial which suggests that RSAC ratings wouldn't stand up to the scrutiny of the Wal-Mart management.

Johnny L. Wilson
Editor-in-Chief,
Computer Gaming World

EDITOR ALEX DUNNE RESPONDS:
Rightfully admonished. Had I only dived deeper into the RSAC web site (www.rsac.org), I would have uncovered an old press release stating that Wal-Mart was one of the first on board to support the RSAC ratings. While I did not attempt to contact RSAC, my two messages with Wal-Mart itself were not returned, so I never got the opportunity to talk directly to store representatives. Apologies to you and the other RSAC committee members.

INDUSTRY WATCH

by Alex Dunne

ROBERTS DIRECTS WING COMMANDER MOVIE. Chris Roberts, who cut his teeth behind the camera during production of the video segments of WING COMMANDER 3 and 4, steps behind the camera again this month as the director of the upcoming WING COMMANDER movie. This is Roberts' debut at the helm of a feature-length film. The $27 million movie has been picked up by Twentieth Century Fox for distribution in the US and UK, and Roberts' own Digital Anvil is going to create the digital imagery for the film. No word yet on a release date or who appears in the movie.

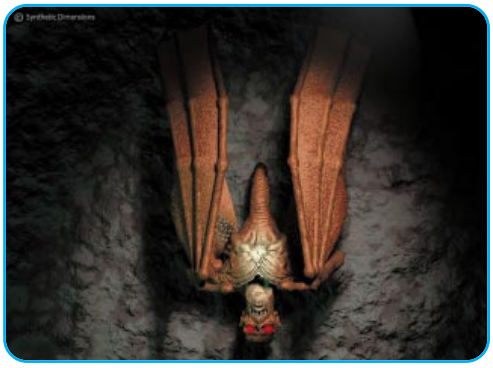
BLIZZARD SAYS BYE-BYE TO COMMERCIAL NETWORKS. With the intense popularity of Blizzard's Battle.net, the company has announced that it it will not license future titles such as STARCRAFT out to third-party networks such as Mpath and TEN, opting instead to host them exclusively on their own free, proprietary network. The company also announced that the site — which recently reached the 1.4 millionth unique user mark — will begin serving paid banner advertisements.

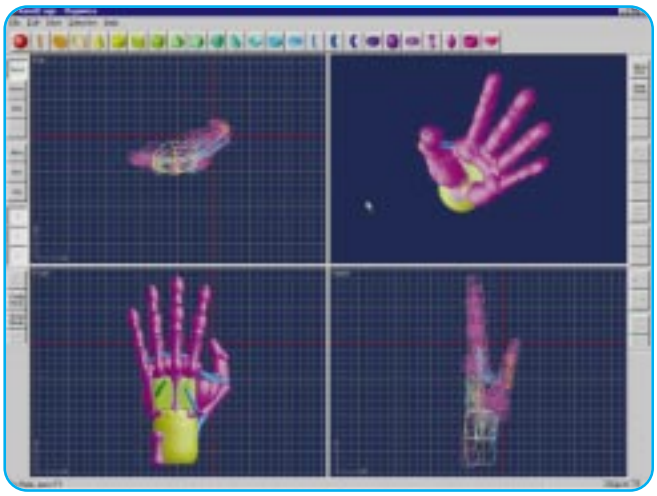
PGL SIGNING SPONSORS. TEN's Professional Gamers League (PGL) has lined up over $2 million in sponsorships so far (including Levi Strauss's Dockers khakis), and this thing looks like it might have legs. Just as some game developers are reaching cult figure status outside of our industry, the PGL has a good shot at turning highly-ranked players into recognizable figures in the mainstream.

Organica

IMPULSE has announced the completion of Organica, a new 3D modeling program that supports 3D object file formats such as Imagine, LightWave, and 3D Studio MAX.

In Organica, users build objects in a method based loosely on the metaball concept. The program gives you 25 different magic blocks that you can put together, bend, taper, twist, shear, or resize to meet your needs. Running in real time, the product lets you put a high-quality 3D object together quickly. The images at right were modeled in Organica by UK-based developer Synthetic Dimensions for their upcoming game.





Organica runs on Windows 95/NT, and a MacOS version is scheduled for release in January 1998. The program has a suggested retail price of $299.

■ Impulse Inc.
Minneapolis, MN
(612) 425-0557
www.coolfun.com

CodeWarrior Professional 2

METROWERKS has just released CodeWarrior Professional 2, the newest version of the company's line of programming tools that combines Windows- and MacOS-hosted desktop tools into a unified software development package.

CodeWarrior Professional 2 combines a project manager, a source-code editor, and a multilanguage code browser together with compilers and linkers. It supports development for a variety of target processors and operating systems using plug-in compilers and linkers from third-party vendors and other CodeWarrior products. New features include: project files that are interchangeable between Windows and MacOS, the ability to compare two source files and merge changes, version 2.0 of Metrowerks' C/C++ compiler, a "current target" column in the project window, a code browser that works across targets and subprojects, and subproject caching that

ASTS

OF GAME DEVELOPMENT

speeds multiproject builds and supports browsing across subprojects. CodeWarrior Professional 2, like all CodeWarrior products, also features the CodeWarrior two-machine source-level debugger. The Windows 95/NT-hosted version also features support for debugging Java in Internet Explorer 4.0.

CodeWarrior Professional 2 is available for MacOS or Windows 95/NT, and is priced at $449.

- Metrowerks Inc.
  Austin, TX
  (512) 873-4700
  www.metrowerks.com

fusion: VOCODE

OPCODE recently released fusion: VOCODE, a cross-platform DSP plug-in designed to enhance digital audio recording by bringing the classic analog vocoder effect onto the desktop.

fusion: VOCODE launches Opcode's new line of DSP plug-ins, called fusion: EFFECTS. The series provides a way to tailor individual sounds and add texture to mixes. VOCODE moves beyond the "hardware box" approach of most plug-ins (software reproductions of traditional functions like reverb, chorus, and others). It includes control features not commonly found on analog boxes, including level, resonance, depth, and mix — in addition to five-band tonal control. VOCODE will also allow you to fuse one sound's personality with anothers. By doing this, distinctive effects can be generated such as guitar talkboxes, robot vocals, and even pulsating rhythm parts derived from sustained chords.

fusion: VOCODE and the fusion: EFFECTS platform currently support plug-in formats including Adobe Premiere, Audiosuite, and DirectX Media. The plug-ins run on both Mac OS and Windows 95. fusion: VOCODE has a suggested retail price of $149.95.

- Opcode Systems Inc.
  Palo Alto, CA
  (650) 865-3333
  www.opcode.com

Shag: Fur

DIGIMATION is shipping Shag: Fur, a new environment plug-in for 3D Studio MAX that adds fur (and even, to some degree, long hair) to an object's surface.

Shag: Fur generates fur with speed without sacrificing realism. It doesn't create geometry for individual hairs, so it works quickly. However, even though no real geometry is generated, the fur can cast and receive shadows and highlights. Other features allow you to control exactly where fur is applied, as well as the density, color, thickness, direction, leaning, and bend of the hairs. Separate texture maps can be used for most of these options to provide complete control. For example, a texture map of a tiger skin can be used for fur color, while a separate map image is used to control where the hairs are thick and thin. Almost all of these functions are animatable, so motion, growth and color changes are all possible.

Shag: Fur works with 3D Studio MAX 1.2 and MAX 2.0, which run on Windows NT. Shag: Fur has a list price of $295.

- Digimation Inc.
  St. Rose, LA
  (504) 468-7898
  www.digimation.com

MARKET REPORTS SIGNAL INDUSTRY HEALTH. Recent market reports by PC Data and InfoTech indicate that both online and shrinkwrap game sales are on the uptick. First, PC Data reported that September's game software sales were up 27.5% over the previous period a year earlier and were growing about about 40% faster than the overall software market compared to the same period in the prior year. Second, InfoTech forecast that worldwide revenue for interactive software publishing would reach $15.8 billion and grow to $26 billion by 2001. InfoTech expects the fastest growth to come in the Internet multiplayer arena, at an estimated compound annual growth rate of more than 70% (to $3.7 billion in revenue) through 2001. But that pales in comparison to their estimate for packaged sales, which InfoTech predicts will reach $22.3 billion that year.

AND I COMPLAIN ABOUT DEADLINES. In conjunction with the launch of the movie Starship Troopers in November, Sony Pictures Entertainment's Imageworks created an VRML-based game for the film's web site (www.starshiptroopers.com). While the game is a just simple 2D obstacle maze, what's impressive is that the game was written, developed, and staged in just eight weeks — by the same team that did the special effects for the movie itself.

JIMMY JOHNSON, VR SPORTS TEAM UP. To help promote their latest title and raise money for charity, VR Sports is donating $1 to the United Way for every copy of JIMMY JOHNSON VR FOOTBALL '98 that's sold. Kudos to VR Sports for this gesture. We here at *Game Developer* are waiting for some publisher to pick up the rights to MIKE DITKA FOOTBALL '98 — you know the one, in which the Saints' AI is hard-coded to lose every game and after which Ditka exclaims "We suck!"

How I Spent My Summer Vacation, or What I Learned While Working on QUAKE 2

A lot of people in the game industry have asked me about the software development processes used at id software, so I thought I'd take the time to write an article about the processes and philosophies used at id software while interspersing my own opinions and reflections on them. While the procedures in place at id are not perfect, they have resulted in the timely shipment of several popular products, so take this for what you will. This column is pretty dry and matter-of-fact — it's more like a laundry list, to be honest — so while it may not be amusing and entertaining, I hope many of you will find it informative. And just to be clear, this is *not* a recipe for success.

This month, I'll be talking about some of the programming methodologies that we've used during QUAKE 2's development. Next month, I plan to discuss the tools that we employ, both software and hardware, and some related issues.

Team Programming

The programming staff at id consists of three programmers: John Carmack, John Cash, and myself. Programming tasks are split into three distinct groups: graphics, game logic, and "glue." The graphics subsystems (OpenGL and software rendering) consist of the actual code used to render a scene and are encapsulated into two .DLLs, REF_GL.DLL and REF_SOFT.DLL. The game logic is also put in a .DLL, GAMEX86.DLL, which handles all game-specific stuff such as monster intelligence, weapon behavior, physics, and power-up effects. Finally, the "glue" code, which consists of window system interaction, input management, sound, CD management, network protocols, and other not-easy-to-categorize crud, is located in the executable, QUAKE2.EXE.

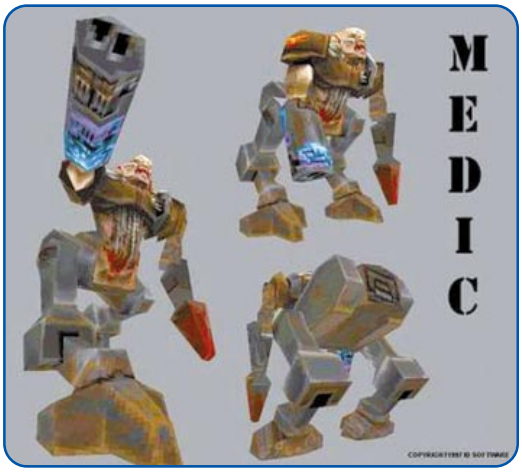
The sweet spot for our team size is working out to be three programmers. The graphics subsystems are my domain; the game logic is John Cash's responsibility; and the glue code is usually modified by any of us. John Carmack is the grand dictator and architect — he modifies broad expanses of all the code at any given time and is responsible for the overall architecture and making sure that the pieces of QUAKE 2 fit together logically and efficiently.

The current triangular hierarchy that we have in place is extremely efficient because John Carmack is the absolute ruler of the programming team. Even though Carmack is the undisputed boss because of his position within id, both Cash and I have extreme respect for him, and it is this respect that allows Carmack to manage the development process effectively. It is far more important to have respect from your employees than arbitrary authority over them. Also, by taking care of implementation details and minutiae, Cash and I allow Carmack to concentrate almost exclusively on large, sweeping architectural issues.

Also, a key to making the team programming approach work, at least for id, is that John Carmack is responsible for both the architecture and the initial implementation of any new technology. This lets him reconcile any unforeseen implementation and design interactions that may have global repercussions within the code base.

Another nice thing about the delegation of responsibilities is that there is

Brian Hook's last Game Developer *column will appear in next month's issue. Tell him how much you'll miss him via e-mail at bwh@wksoftware.com.*



very little adversarial competition between programmers. Neither Cash nor I is presumptuous enough to challenge Carmack's dominance, and since Cash and I work on separate subsystems, our work is complementary instead of competitive in nature. The lines of code ownership are clearly defined and are something with which we're all very comfortable — and we respect each other enough that we don't feel any urge to edit someone else's code. This lets us work in a real team atmosphere, and we manage to avoid the whole "Who's The Man?" jockeying that is so common among computer programmers.

One problem that team programming presents is source control. As ashamed as I am to admit it, id software does *not* use source code control. Right now, this discrepancy is largely the result of expediency. We recognize the need for proper source control, but we have enough of a working system that until source control becomes a crisis, we won't address the problem — especially when we're this close to shipping a product. Our next generation software hopefully will be developed completely within the framework of a large-scale source control system. Personally, however, I use Microsoft Visual SourceSafe on my main workstation simply because I like to have a history of my changes at all times.

Another issue that arises when multiple programmers work together is coding style. It's important not to get wrapped up in religious issues such as tab size, brace and parenthesis placement, or indentation style — you should be able to adjust to any style, even if it irks you. Fighting battles over something as personal as this simply is not worth the effort — make some compromises and move on.

Other coding style issues, however, are worth specifying at the outset of a product's development. Standardization and consistency are very important when working on large projects. Files, data structures, APIs, and variables should have a clean, consistent, and intuitive naming convention so that there is little room for confusion when looking at someone else's code. Static and global variables should be tagged as such, be it by a prefix, suffix, or some other convention. Parameter ordering should be consistent: Is a destination address the first or last parameter? Are prefixes used in **struct** members? Where are global variables declared and under what conditions? Are globals stuffed into a single global structure or just tossed out into the global namespace? How are manifest constants differentiated from **const** declarations? How are directory structures organized?

NIH

For quite some time (over a decade now), computer scientists have been talking about modular software, component software, or "software ICs." The theory is that programmers should be able to purchase a thoroughly debugged and optimized prepackaged software library (who are we kidding?) from some third-party development house and just drop it into a program — voila, instant new features and functionality.

Obviously, there is a difference between that theory and the harsh realities of creating a product that has to ship to real people on a real calendar. Libraries are written by programmers, and programmers are human, and humans make mistakes. Many times, these programmers will even have a different set of priorities than your own.

This is the crux of the problem. The software component that you purchased might look great on paper, but when you drop it into your program and then spend a week looking for a bug that turns out to be a part of your new magic software IC, well, you tend to snap out of your dream world pretty quickly. Anyone who has wanted to firebomb Redmond, Washington, after using Microsoft's DirectX knows what I'm talking about. And when a fix for that bug isn't going to arrive in a timely fashion, you're suddenly in the position of hacking around broken code, a process pleasantly known as "coming up with a workaround." Deal with enough "workarounds," and you'll eventually reach a cross-over point where you realize that you may have been better off if you'd just written the code yourself.

And bugs aren't the only problem you'll encounter — there are performance issues to contend with also. With WINQUAKE, GLQUAKE, and QUAKE 2, we had to work around some pretty serious performance problems in Microsoft's DirectSound. DirectSound works the way it's supposed to, but it's slow enough that you get all teary-eyed remembering the days of Sound Blaster 16 programming under DOS.

Finally, not only do you have to contend with bugs and performance

issues, but there's always the specter of flexibility. That nifty new library might do everything you need *now*, but when you're six months into using that library, and you *must* have a couple new features implemented, and the library owner isn't amenable to adding those features... well, you're in trouble. You now have the option of undoing months of work and rewriting everything from scratch, at which point you've tossed away months of effort, or you forego the extra functionality, which may not be a feasible alternative. And, of course, if you want to port to a development platform or operating system on which the library is not available, you're in deep trouble. You can address many of these problems by licensing the source code to whatever library you're using, but at that point you're in the position of actually learning someone else's code, not to mention maintaining, extending, and debugging it. At some point, you may find that you'd have been better off writing everything yourself from scratch.

Don't get me wrong — I'm not saying that using externally developed libraries is absolutely a bad thing, but some tradeoffs are definitely involved. We have a hard enough time dealing with bugs in our compiler, the Win32 API, Microsoft DirectX, and hardware drivers without adding someone else's code to the mix. So the unofficial policy at id is that we engineer all of our own code unless we absolutely have no choice, such as the necessity of depending on DirectX. It may not be the most effective use of our resources, but it leaves our destiny in our hands, which has a certain warm and fuzzy appeal to it.

Programming Languages

As technology-oriented as id software is perceived, we're actually knuckle-dragging primates when it comes to our programming language of choice. We use good old ANSI C for the majority of our development. Objective-C, a version of C with object-oriented extensions, was the language of choice for tool development back when id was using NextStep. However, during the subsequent move to Windows NT, id was forced to abandon Objective-C in favor of ANSI C for these tasks. We're currently evaluating the performance and robustness of OpenStep for Windows NT, and if it turns out that it doesn't suck, we may switch back to using Objective-C and OpenStep for tool development.

id software still uses ANSI C for its core development; we have several compelling reasons why. We stress portability, and ANSI C is about as portable as a language can get — it's available across a wide range of platforms, and most ANSI C compilers are extremely stable. ANSI C is no longer evolving at a frantic pace, so it's stable in terms of syntax, feature set, and behavior. Mechanisms for interfacing ANSI C with other languages, such as assembler, are well-defined and predictable. Compilers and development tools support ANSI C more than any other language. Finally, ANSI C is a pretty WYSIWYG language — when you look at a chunk of C, you can be reasonably certain what kind of machine code will be generated.

C++, on the other hand, does *not* share these wonderful features. C++ is stuck on top of C using the programming language equivalent of duct tape and twine. It's still evolving at a disturbing rate. It's being designed by a committee. Compilers and tools that support C++ are constantly missing features or incorrectly implementing them, and the language, as a whole, is so large that understanding all of it is nearly impossible. Any given chunk of C++ code, assuming it uses even a small portion of the language, can generate seemingly random assembly code. While you can pick up a book such as Bjarne Stroustrup's *Design and Evolution of C++* to help you understand why C++ is such a screwed up language, it still doesn't address the issue that C++ *is* a screwed up language. It's constantly evolving, getting bigger and uglier, and pretty soon it's going to implode under its own weight.

Seven years ago, I bought Borland Turbo C++ 1.00 the day it was released (yes, I'm that big a geek), and over the course of the ensuing five or six years I used C++ as my only programming language. In that time, I learned most of its weird intricacies, adjusted to them, and accepted them as necessary evils, the price I had to pay for object-oriented programming. When I started working at 3Dfx Interactive, I had to start using ANSI C again because I was developing a programming library, Glide, that needed to be used by a lot of developers, many of whom would not be familiar with C++.

The amazing thing to me was that when I switched back to ANSI C, I was actually *happier* — I discovered a newfound appreciation for ANSI C's simplicity (at least compared to C++). Sure, I lost some nice syntactic sugar, and I ended up missing classes and virtual functions a bit, but I was willing to eschew these niceties in return for simplicity. By using ANSI C, I never had to crack open a reference book,

and I rarely had to work around language quirks. Since then, I've done very little serious C++ programming, and the last time I looked at the spec, it was a completely different language. The language has mutated so much over time that it's become the *Highlander 2* of programming languages — you can see similar themes and names, but somehow the new version screws up so badly that it sullies the name of its parent.

id's use of assembly language has varied over the years. DOOM had very little assembly language in it, but it was primarily targeted at 486-class processors, where scheduling, pipelining, and memory bandwidth issues were not nearly as relevant as on later generations of processors. QUAKE, on the other hand, was targeted at Intel Pentium-class processors, and as a result, there was significant room for hand-coded assembly optimization. It also helped that Michael Abrash, Mr. Assembly Optimization Dude, was working at id then, and could devote large chunks of time to tweaking inner loops.

QUAKE 2 is also targeted at the Pentium, and thus benefits from assembly coding. However, there is a very significant chance that there will be little to no assembly code in post-QUAKE 2 products from id software. There are several reasons for this, the primary one being that hand coding for advanced processors such as the Intel Pentium Pro and Pentium II is often a lost cause. At any given time, a Pentium Pro can have a large amount of non-deterministic internal state that radically affects the efficiency of hand-coded assembly language. With these advanced, superscalar, and superpipelined processors that support features such as speculative, out-of-order execution and branch prediction, it makes far more sense to use CPU-friendly algorithms as opposed to CPU-optimized code.

Optimization

Our optimization rules are quite simple: make sure that what you're optimizing makes a difference, don't optimize before you're done implementing features, and learn to optimize up to the point of diminishing returns but no further. It's amazing how badly our intuition can deceive us when it comes to finding execution hot spots. You'll often look at a program and intuitively label certain areas as definite bottlenecks only to find out, after analytical profiling, that some other heretofore-unknown hot spot is actually consuming all of your execution time. Before diving into "problem" routines, use a profiler such as Intel VTune, Tracepoint HiProf, or Rational Visual Quantify (skip the one in Microsoft Visual C++, it's pretty much useless) to make sure that you are, in fact, going to edit code that makes a difference in your program's overall execution time. QUAKE had a hand-optimized routine responsible for clearing the screen to a flat color. This routine only would be called if the game had texture mapping disabled. Optimizing that clear routine probably wasn't time well spent.

It's easy to get sidetracked into optimizing a chunk of code that you're working on while you're still thinking about it, then tossing away all of that effort when you change your algorithms yet again. Until your overall design is set in stone, it's wise to avoid doing any optimization at all. I had a chunk of code that I was pretty confident wouldn't need to be changed again — it was responsible for transforming the points within an Alias model — but as luck would have it, two months after I optimized that routine, we ended up needing a new feature that required editing the optimized source. Luckily, this wasn't overly traumatic, but if it had been even a bit more complex, it would have consumed far more time than if I had just done all the optimization closer to the end of QUAKE 2's development.

Premature optimization also has another, more sinister, side effect — it makes you reluctant to make major changes to your overall design if it means rendering your previous optimization work irrelevant. A similar adage exists in the world of creative writing — don't keep a bad paragraph just because it has a great sentence. The onset of this mentality can often be very subtle as you start unconsciously weighing the benefits of a potentially better design against the hassle of rewriting your cherished code.

Finally, make sure that you aren't spending more time optimizing a specific chunk of code than is truly necessary. In my experience, the biggest gains during optimization come within the first 25% or so of the time spent — after that the increases are only incre-

mental. There's a cross-over point where the extra effort spent optimizing a piece of code is not reaping commensurate increases in performance. Our blanket policy is that we wait until the final stretch — all features implemented — before doing serious "no retreat, no surrender" optimization work — stuff that requires lots of work and that will end up being wasted time if we change higher-level algorithms and data structures.

Portability

One of the nice benefits of using ANSI C is that our product portability is limited only by our coding discipline. Throughout the development of QUAKE 2 a lot of thought was put into the issue of portability, and by adhering to some general rules, we make our lives a lot simpler when performing a port: segregate OS-specific code from the rest of the code base, always maintain a C-only version of your code, avoid compiler and operating system dependencies, and watch for endianess assumptions.

The first step in writing portable code is recognizing appropriate levels of abstraction and managing your code appropriately. The bulk of our software rendering and sound code only operates on memory addresses — the concepts of DIB sections, DirectDraw, wave sound, and DirectSound aren't known to the actual graphics and sound-generation code, and are instead handled by abstracted glue code. All OS-specific code is sequestered into a set of well defined files — porting QUAKE 2 to an new platform involves touching less than a dozen files, each dealing with some OS-specific subsystem (CD audio, OpenGL, software rendering, window system management, input handling, sound output, and OS-specific utility routines, such as time routines). By doing abstractions at this level, we can greatly minimize the number of conditional compilation directives in our main code body. As a matter of fact, the statement `#ifdef WIN32` only occurs twice in all of our OpenGL code, and it doesn't occur at all in the software rendering subsystem.

We always keep up-to-date C-only versions of our assembly code, both to assist in porting and also because it makes debugging the assembly code extremely easy. It's really easy to forget to maintain your C-only paths, but it does pay off the moment you try to get things up and running on another operating system or CPU.

It's easy to get sucked into compiler and operating system dependencies — for example, utilizing a compiler's `#pragma` directives, or maybe an operating system's message box or memory management functions. Unfortunately, `#pragma` directives are extremely useful for a lot of reasons, and it's easy to forget to bracket them with the appropriate preprocessor directives. The easiest way to avoid operating system dependencies in your main body of code is to make sure that you're not including any OS-specific header files (for example, WINDOWS.H) — the compiler should complain when you start making calls that it doesn't recognize, such as `MessageBox` or `VirtualAlloc`.

Bugs related to CPU endianess can be pretty hard to find, so the best way to prevent them is to recognize code that's doing bad things — for example, casting between different size types. A beneficial side effect of porting to many platforms is that it also forces you to write much cleaner and more robust code, since hidden bugs in your code may only manifest themselves on another compiler, operating system, or CPU. You derive a feeling of confidence from knowing that your code has compiled and run successfully across a wide range of operating systems, compilers, and CPUs. We had a divide-by-zero bug that only generated an exception on the DEC Alpha processor, and it would have been a very long time before this bug was detected in our code under an Intel x86 processor.

However, there are some pretty solid business reasons *not* to port a game to multiple platforms. With QUAKE 2 we plan on supporting Win32 (x86), Win32 (DEC Alpha), Linux (various CPUs), SGI Irix (MIPS), and Rhapsody (x86 and PowerPC). Our publisher will likely receive a disproportionately higher number of support calls for the non-Intel/non-Win32 versions of QUAKE 2 if we release these versions on the QUAKE 2 CD. Couple this with the fact that ports will probably account for less than 3% of our overall revenue, and the argument for supporting a plethora of system architectures becomes pretty flimsy.

We do it anyway, though, because it's cool.

In the end, porting to alternate architectures simply doesn't make very good business sense for most game companies. We do our ports for only three simple reasons: it's easy to do, we like seeing our games played by as many people as possible, and we gain the intangible benefit of having extremely loyal consumers on systems with poor mainstream support.

Stay Tuned

Because the story is just so darn big, I've had to save some for later. Tune in to this space next month for more on QUAKE 2. I'll be explaining the tool choices that we made, complimenting some vendors, and denigrating a few more. ■

Part 1: Real-time 3D Art Tools Need Help!

Call me cranky, but I have something to say: we artists, especially low-polygon modelers, have tools that are holding us back... and they aren't going to get better without our help. In a way, that's to be expected; the job title "real-time 3D artist" (RT3D) is only a couple of years old, so of course there isn't an array of perfect tools out there yet.

The future is what scares me. Even when tool makers listen, we artists aren't talking. Our tools aren't going to get any better unless professional artists explain their needs in a way that tool makers can understand. I'm only one artist in the industry. Tool makers need to hear from other artists (or else we'll get a Josh-centric tool — or more likely, the brush-off). So, RT3D artists, tell me what's wrong with your tools (column@vectorg.com), and I'll publicize the gist of it. I'll start with my main issues.

3D ART TOOLS AREN'T VERSATILE ENOUGH. Artists have a reputation as being independent (if not downright weird), and that's critical to art. RT3D artists' tools are forcing us all to work in very similar patterns. That's wrong. Artists need many different ways of doing the same thing — especially the small variations on the same basic idea. In writing, it's obviously critical; imagine how lame writing would be if an author could only use "good" instead of great, wonderful, terrific, excellent, amazing, superb, awesome, or stunning.

I'm talking about synonyms of features — multiple similar features. "Rich feature set" refers to a variety of features; that's not what I mean. For example, AutoCAD's hardly a world-standard for art tool excellence, but when AutoCAD users pick a point, their options are plentiful. They can use over a dozen object snaps, including weird ones such as "tangent" and "perpendicular," grid snaps, constraint to an axis or plane, offset from an existing point (in polar or planar coordinate systems), composition of the point from the coordinates of several others ("X from point B, but YZ from point C"), or completely custom functions via the built-in macro language.

Most technical 3D modeling software (SDRC I-DEAS, AutoCAD, PATRAN) lets the user choose a symbol to represent a point: a cross, a dot, a box, or a text label. This is unusual in 3D art tools; we get only one choice for vertex viewing. (Nitpicky? On-screen image matters a lot to artists. For example, little "+" signs can easily draw over short edges, obscuring subtle detail in the mesh.) Even Windows 95 offers Control-S, Alt-F-S, mouse clicks, and combinations of menu shortcut key presses and mouse clicks. If you love mouse and toolbar approaches, you should have those options.

Artists have a reputation of being independent (if not downright weird) and that's partly what makes great art. We need tools with large 'vocabulary' to express it.

Clearly, tool developers face a major challenge. But they would be wise to address a few specific problems. For instance, developers have long lists of planned improvements to their products, and sexy, major features make for more impressive advertising copy (or so they think). I suspect they would consider my set of feature synonyms a small "core functionality" improvement that would take significant development effort, and it'd get delayed endlessly.

Of course, these multiple interface paths to the same database edit start making developers' flow charts look like spider webs. This flexibility is where the underlying architecture of the software shows. If the software wasn't well designed and cleanly coded, it's very difficult for the tool maker to avoid bugs when implementing this type of new feature.

It's damned hard to present the user with a ton of different methods without overwhelming an already-complicated UI, and I think most tool developers are overly focused on "easy to use" — they're assuming artists won't bother to learn their tool if they add too many features. Unfortunately, 3D art tools often take the worst of the Windows UI and then strip out the versatility that makes it valuable. In Microsoft Word, darn near every function in the whole program is accessible via the keyboard, either through

Josh White runs Vector Graphics, a real-time 3D art production company. He wrote Designing 3D Graphics (*Wiley Computer Publishing, 1996), he has spoken at the CGDC and cofounded the CGA, an open association of computer game artists. You can reach him at josh@vectorg.com.*

menu shortcuts or with customizable shortcut keys. For example, there's no direct keyboard shortcut to change the font of a style, but you can use the menu shortcuts (Alt-O,S,M,O,F). This sequence is hardly intuitive, but if you forget it, you can look at the menus for a reminder. If you're a fast typist, six keystrokes are much faster and more reliable than six mouse point and clicks.

When paying customers need their product to work differently, tool makers listen closely.

In most 3D art software UIs, the majority of the powerful editing commands are only accessible through mouse-clicked toolbars. This requires accurate, repetitious mouse movement as you navigate through roll-ups and drop-down lists. Witness the lame "keyboard input" windows where you can type in coordinates, but you have to mouse over and click exactly on the "+" roll-up button use it. It's better than no keyboard input at all, but it's hardly an alternative to mouse movement.

IT'S MY WORKSPACE — LET ME ARRANGE IT! The hundreds of buttons packed onto the screen are impressive at trade shows, and for new users, they're convenient reminders of a function's existence. For experienced users, however, they waste desktop space. Who uses toolbar buttons for Cut/Copy/Paste? Naturally, you want to remove them to save space (as well as declutter your work area), but in most mainstream 3D art software with nonstandard toolbars, this is impossible.

To me, an easy-to-learn but inflexible interface shows that the tool maker doubts its users' commitment to its tool. The tool is hard-wired to be E-Z, which means it's meant only for users who aren't going to use it for long — a self-fulfilling prophecy.

INNOVATE! Incremental improvements are good, but tool makers also need to go out on a limb, design-wise. There's no really smooth, easy-to-use interface for 3D modeling — until one exists, any convergence in software is premature (and dangerous to market leaders, since it makes space for upstarts to stage revolutions). Artists need to be a part of this design process. Unless you're 100% happy with the tools you're using, remember to keep an open mind about new tools. One practical way to do this is by keeping a wish list. Whenever you find yourself gnashing your teeth about some 3D modeling problem, pop up Notepad and jot down the annoyance.

Then tell people who care. Attend at least one trade show and when the sales people attack you, attack right back with the wish list. You may not find the perfect tool, but by asking for specific features, you're giving feedback to the tool makers... and before long, you'll find yourself confronted with a dozen different paradigms for 3D modeling.

BORROW FEATURES. I want tool makers to incorporate (and improve on) competitors' good ideas shamelessly. The same applies for 3D modeling in other industries. Take a look at custom 3D software for aerospace, training simulation, mechanical engineering, GIS, medical, and other specialized industries. Skim off the few great ideas. We'll thank you for it.

COMPATIBILITY. There are two forms of compatibility I think about: data transfer and backwards-compatibility (also called legacy issues). The first is critical, not very sexy, and very difficult (no wonder so few tools do a good job of it), but without a robust way to get data in and out of a modeling tool, it's useless. Tool makers need to improve this by agreeing on a common file format (for example, VRML for RT3D) and writing really solid input/output functions for it. If you make it a plug-in, distribute source code so that developers can understand what you did.

After a few years of gradual improvements, 3D art tools are often rebuilt entirely, retaining only the name of the last product. The transition is tough, both for tool makers (who risk their user base) and users (who feel

Research & Development: Break Down the Wall

The top-down structure of large software companies operates on the architect/contractor model. The product architect assembles customer surveys into a clean new product design, which is then thrown over the wall to a complacent, obedient programming department that never sees the actual customer (or even knows how to use the product it constructs).

The defending argument goes, "half-finished 'good ideas' aren't going to help a project this size." The only true gods are the schedule and the specifications. These companies don't want freewheeling cowboy coders who actually have used the product and can form their own opinions. A plague of feature creep (the development of weird, sometimes brilliant, new features that weren't in the spec) will curse their clean environments and incite revolution among the other coders. The feeling is mutual: Creative developers are scared away because developing other people's design is not a creative job.

This leaves behind a group of calm, balanced professionals who are extreme team players, but don't ever get to create their own designs. If you've ever met the staff of a really successful project, you know why I find this upsetting — they're talented and organized, but not calm, and they definitely build their own designs.

It's true that feature creep hurts timely production, but the alternatives are worse. Complacent, user-ignorant programmers turn out bland, bloated products that don't meet the needs of their users. The teams are missing that intense focus, that deep desire that motivates in a way money never can.

To fix this, we don't need a revolution, but we do need change. One way is get face-to-face with users. For example, celebrate beta ship dates with field trips to customer sites. Regularly sit down with users and watch them work, offering suggestions and getting input. Key developers can adopt a client and work onsite once a month. If the wall between Research and Development falls, then tool programmers will understand their users, and the software will improve mightily.

abandoned). Still, I'm impressed with tool makers who have the guts to throw away old code — it makes for better software. Ideally, tool architecture allows sections to be rebuilt without disturbing the remaining parts.

As a user, I can digest an incremental improvement (that is, tools created from a section-by-section rebuild) much more easily than a complete revolution every year, but this incremental approach usually introduces new bugs galore. Once old code is thrown away, I think most tool makers do a good job of compromising between new functionality and old working methods.

ENGINEERS, USE YOUR OWN TOOLS. If you're a professional full-time tool programmer, your company should be begging you to use your tool as the customer would. I'm always amazed to meet tool developers who don't. Lots of programmers grumpily call this "marketing" and think it's not their problem. I say it's the "R" in R&D, and it's critical to making killer tools. See "Research & Development: Break Down the Wall."

ADVICE TO ARTISTS. Artists, tool makers need your help. We artists are too quiet. We aren't asking for better than what we have. This is bad because professional tool makers will *never* be truly clued in to our needs. Without our input, they will assume they're doing fine, and we'll keep getting mediocre tools.

Though you probably can't guess it from this column, I'm very grateful to tool makers for providing me anything that makes my job easier. My way of thanking them is to tell them all the problems the tool has. Rude? Actually, that input (and, of course, payment for the tool) is the most valuable thing a user can offer a tool maker. When paying customers need their product to work differently, tool makers listen closely.

If you do RT3D modeling with general-purpose 3D tools, tool makers need to know how you use their software — they probably think you render movies with it. Even tool makers who are specialized in their attempts to make tools for RT3D need input badly — a lot of their previous customers were building from blueprints, not pencil sketches.

If you use 3D software regularly, you should write the people who make it and tell them what you think. E-mail them, call them, visit trade shows and tell them in person. If you're doing something new such as RT3D, the squeaky wheel gets the grease — and we seriously need some grease here.

OK, flame off. Let's actually build that human character we started last time.

Part II: Low-polygon Character Modeling

If you've joined us from last month where we designed a RT3D character, you'll know that we've got 500 textured triangles with which to make our

FIGURE 1. *Character design sketch.*

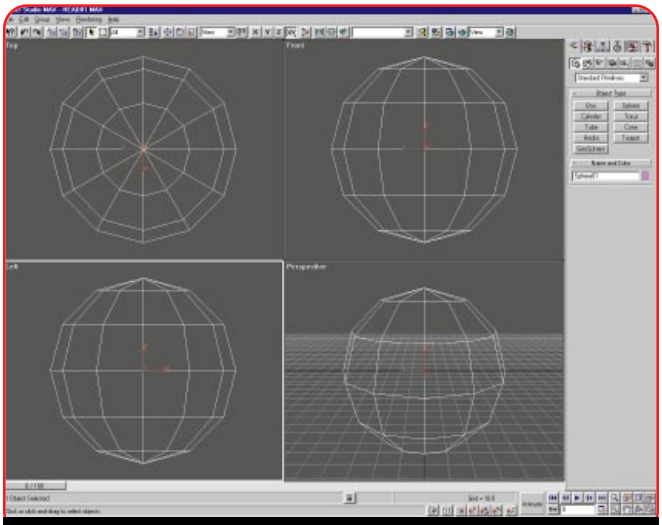


FIGURE 2. *Basic sphere for head modeling.*

character come alive. We also have a clearly defined character, shown in Figure 1, and enough specifics about our environment that we can actually build a model.

Last month, we decided on our approach (including mapping methods and face counts on a per-part basis), identified important areas, and divided up our budgets accordingly. So our task at hand is building a textured 3D model from our sketch and within our face count budget. Here's how we'll tackle it:

- Create the rough geometry. We'll walk through building the head step by step.
- Tune the 3D model to perfection.
- Paint textures on each part.
- Map the parts together into a complete model.

After that, we'll get into animation and revisions. We'll link the parts into a hierarchy, place pivots, and adjust joint designs as necessary. Finally, we'll show our work in the application and do revisions until the model is approved and completed.

FIRST-PASS MODELING. Though we usually start with existing geometry and modify it, if we have to create geometry from thin air, we start with low-polygon primitives and move vertices, divide edges, and weld vertices in an iterative loop. Note that the illustrations here show complete heads, but we usually erase half of the geometry before we start rough modeling, then mirror the existing geometry occasionally to see how it really looks. After we've taken a look, we erase the mirrored half and keep working. Once we're done, we mirror the geometry one last time and weld up unnecessary vertices at the centerline.

Start with a simple sphere with the approximately correct face count, as shown in Figure 2. In a top view, this one has 12 pie slices and four height-lines between the poles, using 96 faces. Now we'll mush this geometry around until it looks like a face. That's easier said than done. Start by looking at the source art carefully and forming a profile along the edge of the sphere in the Left view, as shown in Figure 3. We do this by moving vertices, dividing a few edges as necessary to get the unique features of the profile. Next, we rotate the view and keep moving vertices, turning and dividing edges to define major features. Push a few vertices in for an eye socket, which also forms a bridge for the nose (Figure 4). From there, work the cheekbones a bit and form the chin. For the chin we can change the lowest height-line of vertices to a semi-diagonal edge of the chin. That looks better, but it leaves the underchin/neck area hurting. We'll fix it by dividing a couple of edges, which creates new vertices in approximately the right area. Move those new vertices to define the jaw/neck intersection, turning any edges that connect directly below the chin so those vertices define the corner in profile (Left view). It should now look something like Figure 5.

HEAD COVERAGE. As we look at the sketch, we run into a typical situation: we forgot to plan for accessories. How should we handle the hat in the sketch? Being conscientious artists, we stop modeling and get back into planning mode for a minute. If we build the hat as part of the head (as long as the character doesn't ever need to bare his head), we'll have better performance and less

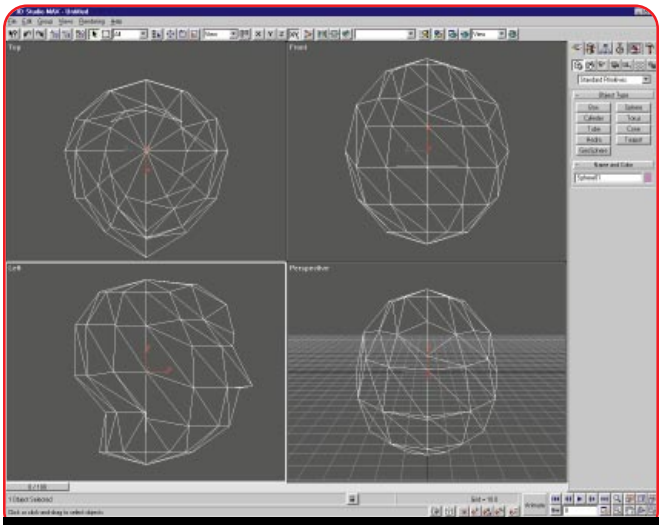


FIGURE 3. *Head profile defined in Left view.*

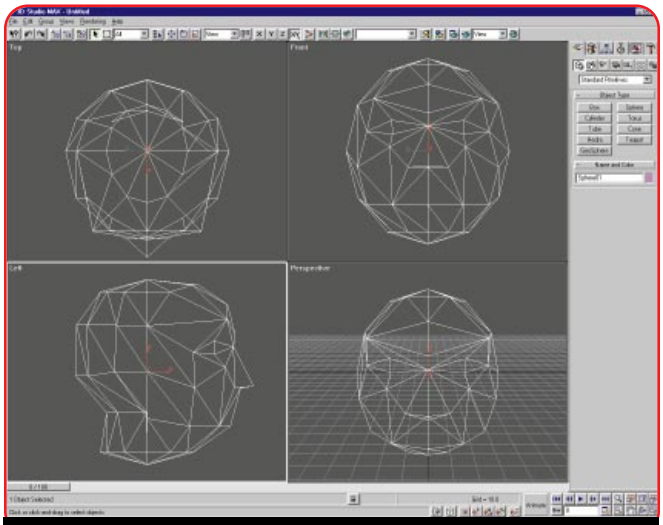


FIGURE 4. *Eye socket formed.*

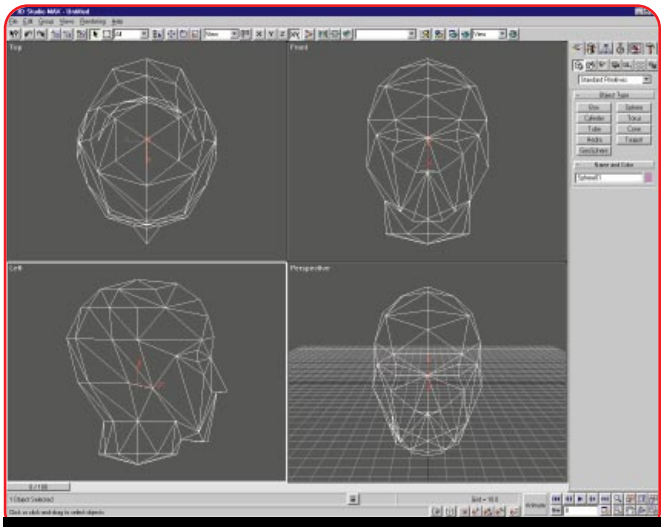


FIGURE 5. *Chin, cheeks, and nose formed.*

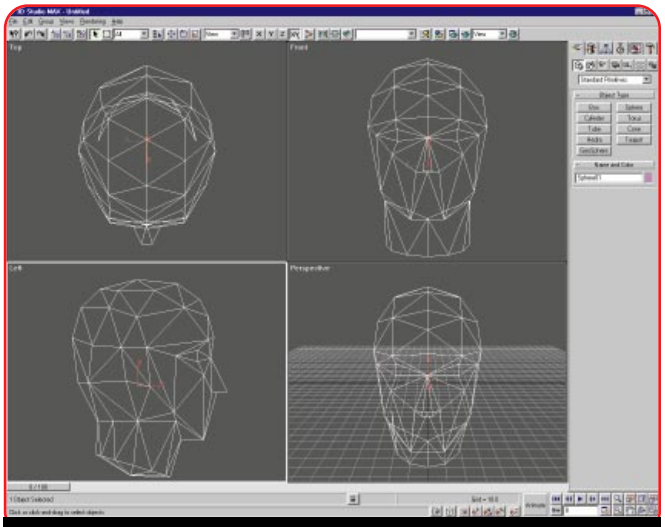


FIGURE 6. *Rough model completed.*

work. After consulting with the art director and game designer, our decision is to model the hair and hat as part of the head.

To build the hat, we grab the top of the head's vertices, move them so they form a diagonal line in the Left view, and scale them a little larger. Then we select a row of four edges near the hairline and extrude them out to form the bill of the cap. Note that these edges aren't perfectly horizontal; they curve around the face somewhat. This is clearly visible in the side view where we see that the profile of the bill is somewhat bulky. This gives us a nice thick-looking bill, even though we didn't spend faces modeling the thickness of the bill.

Once we've got the head roughed out (Figure 6), we mirror it, stand back, and compare it to our thumb just like real artists. Close one eye and make sure the model doesn't look like your thumb. After you've taken a look, erase that mirrored half — we're not ready for that yet.

MEANWHILE, BACK IN YOUR MIND.... Throughout this mushy geometry editing, we'll constantly be revisiting our design decisions. Specifically, we'll need to review which details need to be 3D geometry, and which can be shown in textures. This is a basic decision that was effectively made early on when we sketched the wireframe outline over the pencil sketch; everything not represented by a vertex is assumed to be in the texture. Now that we have the actual model, we'll want to make sure those decisions still make sense (and change our minds as appropriate). For example, you'll notice that we don't have any geometry for the mouth opening — that's because it will look fine as a texture map. The eye sockets, on the other hand, need some 3D depth to look good.

We also need to remember to keep material boundaries represented in geometry. Though it's not an issue for this example, we often need to show expressions on our character. This is commonly done by swapping facial textures at run-time. In that case, we'd do two things:

1) Separate the hat and hair textures from the facial texture. Why? So we don't waste texture memory duplicating the hat and hair images in each facial expression texture.
2) Make edges between the animating facial area and the rest of the head. Since each polygon can have only one texture map, we need separate faces to map the facial animation area onto.

SECOND-PASS MODELING: TWEAKING. Now it's time to tune the rough model. This phase is the pleasant pause after the mad thrashing. Calmly and carefully, we hone the model into a fine work of art. Often, artists will create texture maps before doing this second pass on the 3D model. It's a personal thing — either way works, but since we haven't even addressed material boundaries yet, this example will do second-pass modeling before texturing.

At this point, we need to take a more critical look at the overall proportions, adjusting features to more closely match the sketch. For example, this is a good time to scale the width of the head down a bit since heads aren't usually as sphere-shaped as our model. We'll also move vertices in the mouth area so that it looks more like that of an old man; right

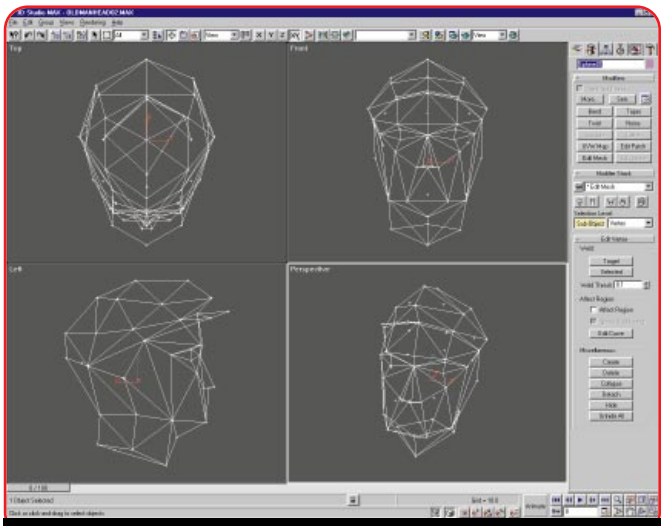


FIGURE 7. *Modeling completed.*

now the chin and cheeks look too taut and young.

We also need to fix the face count — we'll compensate for some of the faces that we added with all those edge divisions by removing faces that aren't defining critical detail. For example, the base of the neck has far too many vertices, especially since it will be hidden inside the torso. We'll get rid of about half of them now.

Simple Characters: How Low Can You Go?

Meet Scared Sam, the artist who builds 100,000-face human models for TV commercials. Sam believes that it's impossible to build any kind of human model with only 500 faces. "Hah!" we reply, puffing out our chests. "We could build a human in only 100 faces! Yes, that's right, a mere 100 faces. Granted, it will kind of suck, but it will be a recognizable human figure." "No way." says Sam. "Prove it, braggart." And so we do.

First, we use triangular cross-sections for the arms and legs. That means we'll have six triangles per straight section. The face counts will be:

two sections (forearm and biceps)	12 faces
cap the end of the limb	1 face
joints	4 faces
per limb:	**17 faces**
Total, 4 limbs:	**68 faces**

That leaves us a luxurious 32 faces to build a head and torso.

The concept for the torso is simple. Start with the connection triangles where the four limbs connect and join them with as few faces as possible. We may notice that the connection triangle's shape has changed some from the one on the limb. This keeps the torso's shape reasonable. This kind of flexing is what sketching is for; we shouldn't feel bound by our previous sketches if they constrain the rest of the model horribly.

This torso design uses 15 faces. With the limbs, we've now used 83 faces, leaving a paltry 17 faces for the head. That's not enough — we'll have to go over our polygon budget a bit.

The hardest part is the head. Here's a synopsis of how to built it: Start with an uncapped extruded pentagon shape. Twist the shape so that the top and bottom pairs of five vertices aren't aligned. Scale the top vertices around their local axis about 120%. Create five new faces that cap the bottom end. Collapse one of the lower edges, creating a four-sided bottom connected to a five-sided top. The long edge that was formed by the collapse is the nape of the neck. Now build a five-sided pyramid to cap the top of the pentagon. Divide the edge in the middle of the forehead. Move this new vertex and the pyramid peak vertex until the head is a little rounder looking.

Scared Sam is impressed. We used 20 faces in this head model, which means we have a 103-face human. Yeah, it's ugly as sin, but it's also surprisingly useful. How else would you make a 50-person angry-mob scene? And And why spend any more faces than necessary on a six-pixel-tall LOD model?

Let's revisit the design and see if we're on track. Specific steps during second-pass modeling are simply iterations of the same types of commands we used in rough modeling, except that the changes are more subtle. Instead of stomping an eye socket out a smooth sphere, we'll be slightly adjusting the depth of the socket. The finished geometry should look something like Figure 7.

TEXTURE MAP CREATION. Creating textures from the pencil sketch is relatively straightforward 2D artwork — essentially, we want to paint full-color, detailed versions of the pencil sketch. There aren't many technical issues that are unique to character texture creation, but let's go through the process.

We could paint the entire front view in a single texture just like the sketch, but this wastes precious texture memory in white space around the arms. So we'll create one texture for each body part. This approach also allows us to use unique (nonsymmetrical) texture for the torso, yet re-use the arm and leg textures on the left and right sides.

Good lighting is critical to good textures. Keep in mind, however, that creating lights in the textures before we have the environment can be a dangerous step to take. If you photograph a red cotton shirt under spotlights and then paste it over a dimly lit room, the shirt will look like shiny red plastic because the white highlights in the photo won't match the room's lighting. This is especially true when the image is so tiny that you can't see the threads in the cloth. Often, the easiest and most versatile solution is to avoid hot spots (bright or white reflections) on non-shiny materials — skin, denim, cotton, and the like should be pretty uniformly lit.

Contributors

Lisa Washburn is the lead RT3D artist at Vector Graphics. With her background in fine art, she uses sculpturing skills as well as her 3D modeling abilities to do her magic.

Lynell Jinks is a professional artist for Vector Graphics. He created the pencil sketches and textures shown in this column. His talent in 2D character artwork spans natural media as well as Photoshop texture and image creation.

FIGURE 8. *Head texture.*

Related to lighting, but not the same, is the ambient light level. It sounds obvious that the basic color of the textures should be established and consistent, but it's easy to get wrong. One technique for preventing confusion is to agree on a single RGB value as a background color. The textures should be visible against it, which essentially means that it should be somewhere near an average color.

Enough preaching to the choir on how to draw textures. I'm sure you've already concluded that we should establish lighting levels and methods, and plan on trying out a few different highlight/contrast schemes until we get good-looking cloth and skin.

Creating cylindrical textures from scratch is difficult because it's harder to imagine how the texture will look on the object. Placing highlights isn't as intuitive. It's much easier to work from existing textures, especially for alignment of faces. If you can find a Cyberware scan of somebody's head, this is a great starting point for painting cylindrical maps of faces (Figure 8).

The Rest

We'll build the rest of the body next month, then we'll assemble the parts into a hierarchy, place pivots, and adjust joint designs as necessary. We'll also prepare simple hand-keyframed animations, and walk through steps of applying motion capture data onto the model.

Feedback is always welcome. E-mail column@vectorg.com and let me know what you thought. ■

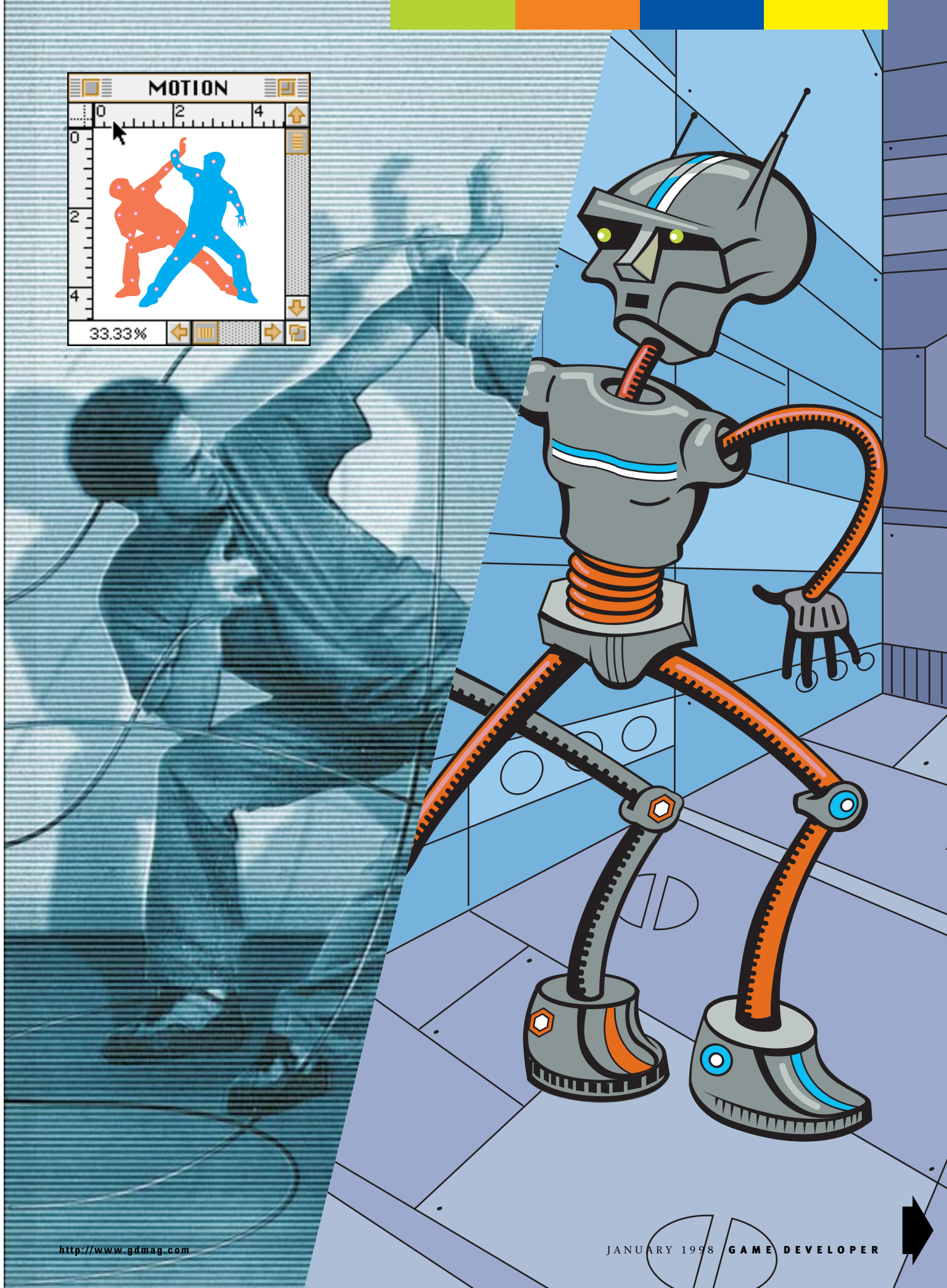
WORKING WITH MOTION CAPTURE FILE FORMATS

BY JEFF LANDER

Boy am I glad that 3D acceleration hardware is here to stay. I'm sure you all feel as liberated as I do by not having to write all that basic polygon stuff. Clipping, sorting, and drawing pixel-by-pixel is about as dull as 3D programming gets. Now I have all this great hardware to do the mind-numbingly dull texture mapping and Z-buffering for me. I also have the render speed and horsepower to do some really interesting stuff. What am I going to do with all this spare time? Really cool real-time 3D characters!

Sure, we've all seen real-time 3D characters. We've even seen real-time 3D characters with a restricted use of animation. However, there have been so many limitations,

Jeff Lander is a Digital Evolutionist at Darwin 3D, where he crafts technology for the future of gaming, entertainment, and network communication. He can be reached at jeffl@darwin3d.com.


MOTION
0
2
4
0
2
4
33.33%

and we all want so much more. We want realism, but how do we go about fulfilling these sick desires? The answer: motion capture.

Motion Capture

Let's face it, motion capture is hot. In the last couple of years, motion capture has spread everywhere — from movies to television commercials, from sports titles to action games, even to click-and-explore adventures. Publishers are climbing all over each other trying to get the words "Motion-Captured 3D Characters" on their boxes. A lot of hype has been loaded onto those words, often to the player's disappointment. As usual, our expectations exceed what the technology can truly deliver. But we're getting so much closer; we have new ripping hardware and the experience from past-generation motion capture downfalls. Yet the desire for more keeps increasing.

The hype has gotten so over-the-top that for the last couple of years, I've been threatening to put out VIRTUA HANGMAN as a demo at E3. I can just see it, these realistic real-time 3D characters marching up to the gallows as you relentlessly guess letters. If we want to be cliché, we could even have a 3D character turning the letters. Now *that* would be an excessive use of technology.

While I wouldn't consider using motion capture for characters better suited to traditional keyframing, motion capture technology clearly has a place in game development. Luckily, the techniques needed for programmers to apply motion capture data to real-time characters work equally well with any type of animation data, be it keyframed, motion captured, or animated through procedural dynamics.

The Need

Let's imagine a scenario in which your brilliant producers have assigned you, the programmer, to develop a real-time 3D character-based game. They have charged you with the tasks of designing the game engine and creating the production pathway. For a variety of design, budgetary, and staffing reasons, you've decided to use motion capture to supply the bulk of your animation data.

Your first task is to decide where you're going to get this data. It doesn't really matter whether you have your own capture setup or a service bureau is doing it for you — plan on plenty of cleanup time. Motion capture is not simple. The data needs quite a bit of massaging to get it ready for the game, and you can get in trouble by underestimating the amount of post-production work the data needs. You also need to be aware that motion capture data is specific to the hierarchy and body dimensions of the person captured. It's possible, but tricky, to scale this motion to other body types and sizes. However, I would recommend getting all your data from one session with one capture artist. This will make your life much easier in the long run.

Still, as an experienced production company, you won't be burdened with these details because your producers have budgeted the motion capture session correctly. Now you need to decide how you want this data to come to you. Other formats exist, but the Biovision (.BVA/.BVH) formats and the Acclaim Motion format are the big ones, and all the service bureaus and animation packages support these.

Your file format decision depends on your application and engine needs. You can bring these formats into a commercial animation package and export the data from there, but the formats are very compact and easy to use with your own tool set.

Definition of Terms

I'll refer to the character that you apply motion capture data to as a *skeleton*. The skeleton is made up of *bones*. To create the character's look, you attach geometry or weighted mesh vertices to these bones. The attributes that describe the position, orientation, and scale of a bone will be referred to as *channels*. By varying

LISTING 1. *Sample Biovision .BVA file.*

```
Segment:   Hips
Frames:    29
Frame Time: 0.033333
XTRAN      YTRAN      ZTRAN      XROT       YROT       ZROT       XSCALE  YSCALE  ZSCALE
INCHES     INCHES     INCHES     DEGREES    DEGREES    DEGREES    INCHES  INCHES  INCHES
0.000000   34.519684  0.000000   -14.988039 -12.240604 -3.481155  ...
0.102748   34.078739  3.159979   -15.337654 -14.320413 -3.983407  ...
0.260680   33.836613  6.487895   -16.308723 -15.090799 -3.861260  ...
...        REPEATS FOR A TOTAL OF 29 FRAMES
Segment:   Chest
Frames:    29
Frame Time: 0.033333
XTRAN      YTRAN      ZTRAN      XROT       YROT       ZROT       XSCALE  YSCALE  ZSCALE
INCHES     INCHES     INCHES     DEGREES    DEGREES    DEGREES    INCHES  INCHES  INCHES
0.272156   38.993561  -1.199981  -4.022753  -0.411088  1.354611   ...
0.413597   38.542671  1.932666   -4.371263  -0.591130  1.100887   ...
0.560568   38.279800  5.184929   -5.020082  -0.657020  0.768863   ...
...FOR THE REST OF THE SEGMENTS
```

the value in a channel over time, you get animation. These channels are combined into an *animation stream*. These streams can have a variable number of channels within them. Each slice of time is called a *frame*. In most applications, animation data has 30 frames per second, though that's not always the case.

BIOVISION'S .BVA FORMAT. This is probably the easiest file format to handle. It's directly supported by most of the 3D animation packages. Let's take a look at a piece of a .BVA file (Listing 1).

This is as simple as animation data gets. For each bone in the skeleton (or what Biovision calls Segments), there are nine channels of animation. These represent the translation, rotation, and scale values for each bone for each frame. You'll also notice that there is no hierarchy definition. That's because each bone is described in its actual position (translation, rotation, and scale) for each frame. This can lead to problems, but it sure is easy to use.

Figure 1 shows the hierarchy of a sample .BVA file.

In Listing 1, we see that **Hips** as the first bone described. There are 29 frames of animation in the **Hips**. The frame time is described as 0.03333 seconds (per frame), which corresponds to 30 frames per second. Next comes a description of the channels and units used, then the actual channel data. There are 29 lines of nine values, followed by a segment block that describes the next bone, and so on, continuing to the end of the file. That's all there is to it.

FIGURE 1. *.BVA file hierarchy.*

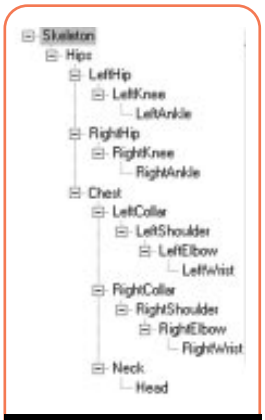


FIGURE 2. *.BVH file hierarchy.*

BIOVISION'S .BVH FORMAT. This format is similar to the .BVA format in many respects. In practice, I know of no off-the-shelf way to import this file format into Alias|Wavefront or Softimage, although Biovision's plug-in, Motion Manager for 3D Studio MAX, reads it. Still, it's an easy-to-read ASCII format that can be useful for importing and storing animation data. Obtaining data in this format should be easy because the format is supported by many motion capture devices and service bureaus.

The .BVH format differs from the .BVA format in several key areas, the most significant of which is that .BVH can store motion for a hierarchical skeleton. This means that the motion of the child bone is directly dependent on the motion of the parent bone. Figure 2 shows a sample .BVH format hierarchy.

In this sample, the bone **Hips** is the root of the skeleton. All other bones are children of the **Hips**. The rotation of the **LeftHip** is added to the rotation and translation of the **Hips**, and so on.

This hierarchy will certainly complicate the game engine's render loop. Why would you want to bother? You can do many more interesting things if your motion is in a hierarchy. Let's take the example of wanting to combine a "walk" motion with a "wave" motion. In the .BVA format, there is no relationship between the **LeftUpArm** and the **Hips**. If we were to apply a different motion to the different bones, nothing would stop them from separating. A motion hierarchy allows you to combine such motions fairly easily. Also, should we ever want to add inverse kinematics or dynamics to the game engine, a hierarchy would make this possible.

Listing 2 shows a fragment of a .BVH file. The word **HIERARCHY** in the first line signifies the start of the skeleton definition section. The first bone that is

LISTING 2. *Sample Biovision .BVH file.*

```
HIERARCHY
ROOT Hips
{
      OFFSET  0.00     0.00    0.00
      CHANNELS 6 Xposition Yposition Zposition Zrotation Xrotation Yrotation
      JOINT LeftHip
      {
            OFFSET    3.430000            0.000000    0.000000
            CHANNELS 3 Zrotation Xrotation Yrotation
            JOINT LeftKnee
            {
                  OFFSET   0.000000    -18.469999    0.000000
                  CHANNELS 3 Zrotation Xrotation Yrotation
                  JOINT LeftAnkle
                  {
                        OFFSET       0.000000   -17.950001   0.000000
                        CHANNELS 3 Zrotation Xrotation Yrotation
                        End Site
                        {
                              OFFSET       0.000000   -3.119999    0.000000
                        }
                  }
            }
      }
...
}
MOTION
Frames:     20
Frame Time: 0.033333
 0.00  39.68    0.00     0.65   ...
...
```

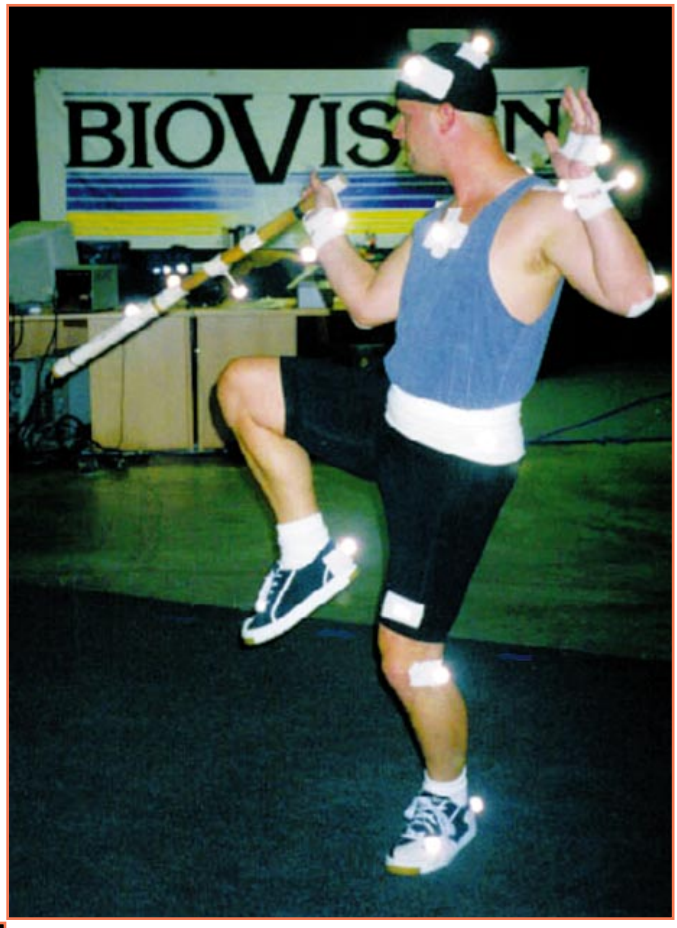
defined is the **ROOT**. This bone is the parent to all other bones in the hierarchy. Each bone in this hierarchy is defined as a **JOINT**. Braces contain the root and each joint. All joints within a set of braces are the children of that parent joint.

Within each braced block is the **OFFSET** and **CHANNELS** definition for that bone (or **JOINT**). The **OFFSET** describes displacement of the root of the bone from its parent. This (x,y,z) coordinate is the world coordinate offset from the parent bone. In the example, the **Hips** bone is located at offset (0,0,0) and the **LeftHip** is 3.43 world units away from the **Hips** in the x axis.

The **CHANNELS** line defines which bone parameters will be animating in the file. The first parameter is the number of channels animated for this bone. Next is a data type for each of these channels. The possible types are: **Xposition**, **Yposition**, **Zposition**, **Xrotation**, **Yrotation**, and **Zrotation**. Note that the scale channels have been dropped in the .BVH format. Normally, only the root bone has any position data — the rest of the bones have only rotational data and rely on the root and the hierarchy for their position. The **CHANNELS** can be in any order. This order defines the sequence in which the operations need to be processed in the playback. For example, in the **LeftAnkle** joint, the order of channels is **Zrotation Xrotation Yrotation**, meaning that the bone is first rotated around the z axis, then the x axis, and finally the y axis. This becomes important when we try to display the data.

The branch of the hierarchy ends with the **End Site** joint. This joint is offset is only useful in determining the length of the last bone.

Following the **HIERARCHY** section is the **MOTION** section. This section actually describes the animation of each bone over time. As in the .BVA format, the first two lines of this section describe the number of frames and the time for each frame. However, unlike the .BVA format, the next lines describe the animation for all the bones at once. In each line in the rest of the **MOTION** section, there is a value for every **CHANNEL** described in the **HIERARCHY** section. For example, if the **HIERARCHY** section describes 56 channels, there will be 56 values on each line of the **MOTION** section. That continues for the total number of frames in the animation.



That's it for the .BVH format. While it's a bit more complex, it gives the programmer designing the engine greater flexibility.

ACCLAIM SKELETON FORMAT. This is the most complicated of the three file formats. It's also the most comprehensive, and supported by most of the 3D animation packages. An Acclaim motion capture file is actually made up of two files; the .ASF, which describes the actual skeleton and its hierarchy, and the .AMC file, which contains the motion data. The separation of these two files has a nice benefit. In a single motion capture session, you can have one .ASF file that describes the skeleton and multiple .AMC motion files. The Acclaim format is such a technical and complex file format that this overview may not provide all the needed information. Documents describing the format in greater detail are available on the *Game Developer* web site (http://www.gdmag.com).

The .ASF file is similar to the **HIERARCHY** section of the .BVH file in many ways. Both files describe the joints and the hierarchy, but the .ASF file extends this a bit. Listing 3 displays a portion of an Acclaim .ASF file.

In this file format, lines beginning with a pound sign (**#**) are ignored. The .ASF file is divided into sections. Each section starts with a keyword preceded by a colon. The section continues until another keyword is reached. The **:version**, **:name**, and **:documentation** section are self-explanatory. The **:units** section describes a definition for all values and units of measure used.

The **:root** section describes the parent of the hierarchy. The **axis** and **order** elements describe the order of operations for the initial offset and root node transformation. The **position** element describes the root translation of the skeleton and the **orientation** element defines the rotation.

The **:bonedata** keyword starts a block that describes all of the remaining bones in the hierarchy. Each bone is delimited by **begin** and **end** statements.

LISTING 3. *Sample Acclaim .ASF file.*

```
:version 1.10
:name BioSkeleton
:units
   mass 1.0
   length 1.0
   angle deg
:documentation
   Data translated and provided by
   BioVision Motion Capture Studios
:root
   axis XYZ
   order TX TY TZ RZ RY RX
   position 0.0 0.0 0.0
   orientation 0.0 0.0 0.0
:bonedata
begin
   id 1
   name hips
   direction 0.000000 1.000000 0.000000
   length 0.000000
   axis 0.00000 0.00000 0.00000 XYZ
   dof rx ry rz
   limits (-180.0 180.0)
      (-180.0 180.0)
      (-180.0 180.0)
end
begin
   id 2
   name hips1
...
end
:hierarchy
begin
   root body_root1
   body_root1 hips
   hips hips1 hips2 hips3
...
end
```

LISTING 4. *Sample .AMC file.*

```
:FULLY-SPECIFIED
:DEGREES
1
root -1.244205 36.710186 7.591899 0.958161 4.190043 -18.282991
hips 0.000000 0.000000 0.000000
chest 15.511776 -2.804996 -0.725314
neck 48.559605 0.000000 0.014236
head -38.332661 1.462782 -1.753684
leftcollar 0.000000 15.958783 0.921166
leftuparm -10.319685 -15.040003 63.091194
leftlowarm -27.769176 -15.856658 8.187016
lefthand 2.601753 -0.217064 -5.543770
rightcollar 0.000000 -8.470076 2.895008
rightuparm 6.496142 9.551583 -57.854118
rightlowarm -26.983490 11.338276 -5.716377
righthand -6.387745 -1.258509 5.876069
leftupleg 23.412262 -5.325913 12.099395
leftlowleg -6.933442 -6.276054 -1.363996
leftfoot -1.877641 4.455667 -6.275022
rightupleg 20.698696 3.189690 -8.377244
rightlowleg 3.445840 -6.717122 2.046032
rightfoot -8.162314 0.687809 9.000264
2
root -4.232432 36.723934 9.596100 -7.051147 1.678117 -7.711937
hips 0.000000 0.000000 0.000000
chest 31.863499 -19.017111 6.490547
...
```

This bone description section is what makes the Acclaim format very useful.

The **id** and **name** elements describe the bone by number or string. The initial rest position of the bone is described by the **direction** vector, and the **length** describes the physical length of the bone. The **axis** parameter describes the global orientation via an axis vector, and the token letters **xyz** describe the order of rotations. Not included in the sample are two optional elements: **bodymass**, which defines the mass of the bone, and **cofmass** which pinpoints the center of mass via a distance along the bone.

The **dof** element describes the degrees of freedom possible in the bone. This is a list of tokens. The possible values are tx, ty, tz, rx, ry, rz, and l. The first of these six define freedom to translate and rotate around the three axes. The last **dof** defines the bone's ability to stretch in length over time. Each of these tokens represents a channel that will be present in the .AMC file in that order. The order of these channel tokens also describes the order of operations in the transformation of the bone.

The **limits** element is very interesting. It describes the limits of the degrees of freedom. It consists of value pairs of either floats or the keyword **inf**, meaning infinite. This information can be useful for setting up an inverse kinematic or dynamic 3D character.

The next section in the .ASF file is **:hierarchy**. Just as it sounds, it describes the hierarchy of the bones declared in the **:bonedata** section. It's a **begin...end** block in which each line is the parent bone followed by its children. From this information, the bones should be connected together in the proper hierarchy. Figure 3 displays the hierarchy in the sample .ASF file.

The .AMC file defines the actual channel animation. Listing 4 contains a sample .AMC fragment. Each frame of animation starts with a line declaring the frame number. Next is the bone animation data, which is comprised of the bone name and data for each channel defined for that bone. This information was defined in the **dof** section of each bone in the .ASF file. The frame sections in the file continue until the end of the file. After the complexity of the .ASF file, the .AMC looks pretty simple.

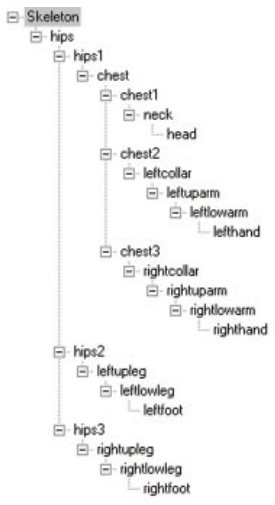


FIGURE 3. *.ASF file hierarchy.*

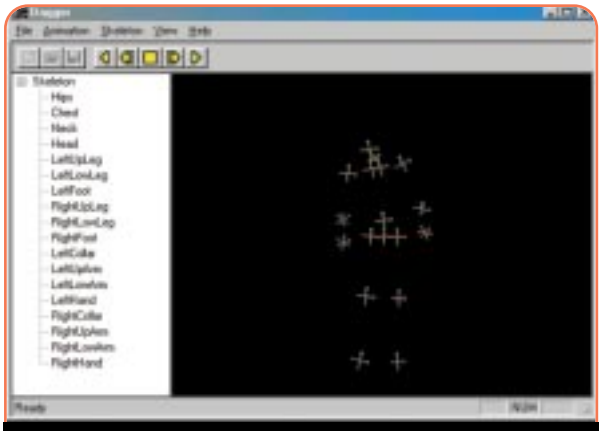


FIGURE 4. *A sample animation in OGLView.*

You should be aware of one important aspect of the Acclaim and .BVH formats. While both formats can store rotations in arbitrary order, both Softimage and Alias|Wavefront expect the order of rotations to be tx, ty, tz, rx, ry, rz. This is important if you plan on going back and forth between the game engine and one of these packages.

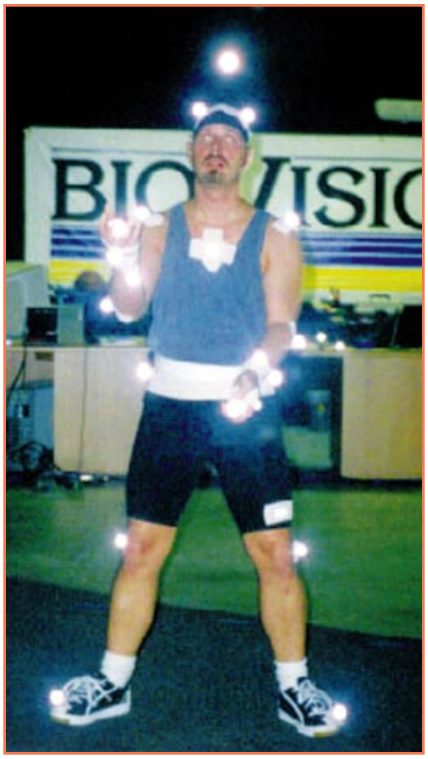
Working with Data

Once you have your data in a format that you're happy with, it's time to start working on it. I've created an application that loads motion capture files of different formats and allows the user to play them back. You can download it from the *Game Developer* web site. When full production kicks in, such tools are very useful for file conversion and formatting. Also, it serves as a good test application to try out new ideas and benchmark code.

I decided to create the motion capture viewer as a MFC OpenGL application — I find it very quick and easy to create tools this way. If you're careful about how you design the tool, much of the code can be used directly in the game engine itself. Figure 4 shows a sample animation that has been loaded into the application.

DATA REPRESENTATION. As we saw from the different file formats, there are several ways to store the animation data from a motion capture session. The most important data is the order of rotations in each stream. You may remember from 3D matrix math that matrix multiplication is noncommutative (see "Inspecting the 3D Pipeline," Casey Muratori, *Game Developer*, February/March 1997, pp.38-41). The order in which you perform these operations is critical to getting the expected result.

Because I wanted my motion capture viewer to support several different file formats, it was important to take operation order into account. I created a series of stream IDs that describe the order of channels in each stream.



Listing 5 shows the stream types that I've needed. These are not all the possibilities, but they are the ones that I've found useful. By creating separate stream types for single operations such as **STREAM_TYPE_TRANS** and **STREAM_TYPE_RXYZ**, I decrease stream size while animating. This operation isn't as important when the animation can fit in RAM, but it becomes critical when you have to stream animation off of a CD-ROM or the Internet.

Next, I created a data structure to represent a bone (Listing 6). I chose to store the transformation information as separate scale, translation, and rotation vectors instead of a global transformation matrix. This made it much easier to handle the different channel types. I also find the rotation values, called Euler angles, easy to understand while debugging. Conversion to and from quaternions for animation or a transformation matrix can also be done easily. Once the data format for a game engine is set, this can be optimized.

Looking at the **primStreamType**, **primStream**, and **primFrameCount** fields, it seems curious that I would want to have a motion stream for each bone. It would certainly be easier to have one animation stream that contains all the data for all the bones in the skeleton. However, this method allows the flexibility to have different stream types per bone. This can be useful because it allows me to attach completely different motions to the individual bones. Imagine a character in a walk cycle. The legs and hips are

LISTING 5. STREAM *definitions from* Skeleton.H.

```
/// STREAM Definitions ////////////////////////////////////////////////////
#define STREAM_TYPE_NONE            0       // NO STREAM APPLIED
#define STREAM_TYPE_SRT             1       // SCALE ROTATION AND TRANSLATION
#define STREAM_TYPE_TRANS           2       // STREAM HAS TRANSLATION (X Y Z) ORDER
#define STREAM_TYPE_RXYZ            4       // ROTATION (RX RY RZ) ORDER
#define STREAM_TYPE_RZXY            8       // ROTATION (RZ RX RY) ORDER
#define STREAM_TYPE_RYZX            16      // ROTATION (RY RZ RX) ORDER
#define STREAM_TYPE_RZYX            32      // ROTATION (RZ RY RX) ORDER
#define STREAM_TYPE_RXZY            64      // ROTATION (RX RZ RY) ORDER
#define STREAM_TYPE_RYXZ            128     // ROTATION (RY RX RZ) ORDER
#define STREAM_TYPE_S               256     // SCALE ONLY
#define STREAM_TYPE_T               512     // TRANSLATION ONLY (X Y Z) ORDER
#define STREAM_TYPE_INTERLEAVED     1024    // THIS DATA STREAM HAS MULTIPLE STREAMS
////////////////////////////////////////////////////////////////////////////
```

affected by the **walk** stream. Suppose I then attach a **wave** stream to the right arm. Now I have a walking and waving character. We can also begin to plan for the possibility of blending animations together to create dynamic motions on the fly.

DISPLAY METHODS. Now that I have all this data loaded, I have to display it in a way that provides the most information possible. In my tool, I chose to represent each bone of the skeleton as an axis. The axes are colored red for x, green for y, and blue for z. An arrow indicates the positive direction in each axis. Since I was using OpenGL to create my motion capture tool, this seemed like a good opportunity to use display lists. Display lists are a method that OpenGL uses to optimize sequences of commands. Listing 7 contains the OpenGL commands that I used to create a simple colored axis.

I also created a hierarchy browser using the **CTreeCtl** class in MFC. This gives a nice visual representation of how the skeleton is laid out. From there, it's easy to add dialog boxes to edit bone settings, a more proper animation control window, andso on.

The animation is all handled via a Windows timer event. This isn't the fastest way to animate a Windows application, but it's plenty fast for this demonstration. There's a very good discussion on animating OpenGL Windows applications in Ron Fosner's book, *OpenGL Programming for Windows 95 and Windows NT*. I recommend this book and the *OpenGL Super Bible* by Wright and Sweet to any Windows OpenGL programmer.

The source code and executable for this application, along with sample motion files and two documents describing the Acclaim file format can be found on the *Game Developer* web site. ■

Acknowledgements

I wish to give a special thanks to those who contributed necessary information and assets: House of Moves (www.moves.com) and Biovision (www.biovision.com) for sample motion files; and Richard Hince of Probe and Richard Barfield of Oxford Metrics Limited for information on the Acclaim Motion file format.

LISTING 6. *Structure definition from* **Skeleton.h.**

```
struct t_Bone
{
     long         id;                    // BONE ID
     char         name[80];              // BONE NAME
     // HIERARCHY INFO
     t_Bone       *parent;               // POINTER TO PARENT BONE
     int          childCnt;              // COUNT OF CHILD BONES
     t_Bone       *children;             // POINTER TO CHILDREN
     // TRANSFORMATION INFO
     tVector      b_scale;               // BASE SCALE FACTORS
     tVector      b_rot;                 // BASE ROTATION FACTORS
     tVector      b_trans;               // BASE TRANSLATION FACTORS
     tVector      scale;                 // CURRENT SCALE FACTORS
     tVector      rot;                   // CURRENT ROTATION FACTORS
     tVector      trans;                 // CURRENT TRANSLATION FACTORS
     // ANIMATION INFO
     DWORD        primStreamType;        // WHAT TYPE OF PRIMARY STREAM IS ATTACHED
     float        *primStream;           // POINTER TO PRIMARY STREAM OF ANIMATION
     float        primFrameCount;        // FRAMES IN PRIMARY STREAM
     float        primCurFrame;          // CURRENT FRAME NUMBER IN STREAM
     ...
     // REST OF STRUCTURE DECLARATION
};
```

LISTING 7. *Display list code from OGLView.CPP*

```
    // CREATE THE DISPLAY LIST FOR AN AXIS WITH ARROWS POINTING IN
    // THE POSITIVE DIRECTION Red = X, Green = Y, Blue = Z
    glNewList(OGL_AXIS_DLIST,GL_COMPILE);
         glBegin(GL_LINES);
                glColor3f(1.0f, 0.0f, 0.0f);        // X AXIS STARTS - COLOR RED
                glVertex3f(-0.2f,  0.0f, 0.0f);
                glVertex3f( 0.2f,  0.0f, 0.0f);
                glVertex3f( 0.2f,  0.0f, 0.0f);     // TOP PIECE OF ARROWHEAD
                glVertex3f( 0.15f,  0.04f, 0.0f);
                glVertex3f( 0.2f,  0.0f, 0.0f);     // BOTTOM PIECE OF ARROWHEAD
                glVertex3f( 0.15f, -0.04f, 0.0f);
                glColor3f(0.0f, 1.0f, 0.0f);        // Y AXIS STARTS - COLOR GREEN
                glVertex3f( 0.0f,  0.2f, 0.0f);
                glVertex3f( 0.0f, -0.2f, 0.0f);
                glVertex3f( 0.0f,  0.2f, 0.0f);     // TOP PIECE OF ARROWHEAD
                glVertex3f( 0.04f,  0.15f, 0.0f);
                glVertex3f( 0.0f,  0.2f, 0.0f);     // BOTTOM PIECE OF ARROWHEAD
                glVertex3f( -0.04f,  0.15f, 0.0f);
                glColor3f(0.0f, 0.0f, 1.0f);        // Z AXIS STARTS - COLOR BLUE
                glVertex3f( 0.0f,  0.0f,  0.2f);
                glVertex3f( 0.0f,  0.0f, -0.2f);
                glVertex3f( 0.0f,  0.0f, 0.2f);     // TOP PIECE OF ARROWHEAD
                glVertex3f( 0.0f,  0.04f, 0.15f);
                glVertex3f( 0.0f, 0.0f, 0.2f);      // BOTTOM PIECE OF ARROWHEAD
                glVertex3f( 0.0f, -0.04f, 0.15f);
         glEnd();
    glEndList();
```


Back

Adding Planning Capabilities to Your Game AI

by Bryan Stout

any complaints about artificial intelligence (AI) in games can be attributed to a single cause: the AI doesn't understand what it's doing. Actions are determined by a combination of mechanistic rules and internal dice rolls, but often there is no explicit means of representing the reasons for particular actions. We need to define these reasons and represent them in a way that the computer can manipulate. If we can do so, then the game's agents — that is, the autonomous entities, be they simple monsters, NPCs, military units, or computer-controlled players — can demonstrate several intelligent capabilities.

- They can act in terms of goals.
- They can carry out long-term plans.
- They can adapt their behavior to situations not planned for by the game developers.

Fortunately, there are means to improve AI along these lines. This article will explore research results in the AI subfield of planning. This research has been going on for over 30 years (a very long time in computer research terms), and its goal has been to develop routines for agents to achieve goals in an environment that the agents themselves can change. Over time, this field has explored the representation of actions and goals, the reasoning about time and causality, and methods for dealing with unknown and dynamic environments.

Such a large field can only be touched upon briefly. The aim of this article is fairly modest: to explore how some of the basic ideas of planning can be used for a game AI. Those who want to explore the subject further should check out the references at the end of this article. While the techniques that I present won't give your game's AI the reasoning skills of a human, they will help you understand how reasoning can translate into action.

Bryan Stout has worked professionally both in applied artificial intelligence and in computer game development. He has lectured on game AI at several conferences, including the Computer Game Developers' Conference, and has been working on a book about game AI. After a hiatus of a few months, he is pleased to announce that he has signed a contract with Morgan Kaufmann, and his book, tentatively titled Adding Intelligence to Computer Games, *will appear in 1999.*

What's in a Plan?

The parts that go into a representation of a plan or action become apparent after some careful analysis. Think about the plans that you make on a regular basis. Where should I go for lunch? Which errands do I have to do on the way home from work? How should I schedule the development of my game? Most plans share common traits.

- A plan has a *purpose* — a goal to reach. Perhaps there are several goals,

LISTING 1. *Sample rules for a plan to get past a monster and locked door to a treasure.*

```
If not sufficiently armed to fight the monster,
          then look for weapons.
If do not have the key to the treasure room door,
          then look for the key.
If significantly wounded,
          then heal self.
If sufficiently armed and healed,
          then attack the monster.
If fighting the monster and seriously wounded
 and the monster is not seriously wounded,
          then flee.
If the monster is dead or gone and have the treasure door key,
          then unlock the treasure room door key and unlock the door.
If the treasure room door is open
          then get the treasure.
```

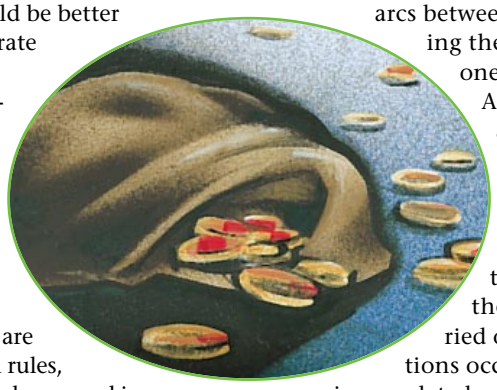

and a meta-goal to achieve them efficiently, but each plan has a purpose.

- A series of *subgoals* must be achieved in order to accomplish a plan. Additionally, subgoals may in turn have their own subgoals.
- A variety of *steps* must be be taken in order to carry out a plan.
- Some steps have *prerequisites*, or conditions, that must be true before they can be taken.
- The steps are taken in order to *achieve* some part of a goal, or a prerequisite of some other step.
- The steps in a plan have an *order*. Some orderings are necessary because of prerequisites; other orderings are arbitrary.
- The steps often use a certain amount of *resources* — including items such as raw materials, money, manpower, fuel, or time — whose presence must be accounted for in order for a plan to succeed.
- Some plans have *conditional steps*, which cause branches in the steps taken depending on some other outcomes.

Implementing the Plans

Of course, the basic elements listed previously can be represented in many ways, from simple to robust.

HARD-CODING PLANS. The simplest way to represent a plan is to write the planning directly into the source code. The quality of the plan being followed depends on the thoroughness of the code as it relates to various incidental situations. The plans can be implicit in the code — the goals result from the interplay between different parts of the code — or perhaps global variables or objects will keep track of goals that have yet to be fulfilled and the progress towards them. It's tempting to hard code this functionality into the game, since it doesn't require designing special data structures. This solution is perhaps the most difficult to maintain, however, because any changes to the AI require modifying and recompiling the code. It would be better if we could separate the knowledge from the reasoning; we need to put the plans in data structures that the planning routines can use.



PRODUCTION RULES. Production rules are condition/action rules, which have often been used in expert systems and other traditional AI applications. A production-rule system loops through all the rules, takes note of which ones apply (that is, which have conditions that are true), and chooses one of them to fire (invoke). The basic condition/ action rule is a very flexible tool and can be adapted to many situations. For planning, the action, or right-hand-side (RHS), of the rule would be the action to take. The condition, or left-hand-side (LHS), would be the prerequisites of the action. The regular rules can be supplemented with variables that keep track of the desired goals and the status of the plans to realize them.

Production rules can be hard-coded or stored in explicit data structures that are defined either in a code module or in an external file. Listing 1 shows some rules (in outline form) that could represent a plan to get a treasure that is protected by a guardian monster and a locked door.

One problem that rule systems must deal with is how to choose between multiple rules that are supposed to fire at the same time. A way to solve this problem is to assign a priority to each rule or to the different conditions that can appear in rules.

Another problem is that if a rule system gets too large, it's inefficient to test all of the rules' conditions every cycle. Fortunately, there are ways to speed up this process. One way is to index all the rules by the clauses in their LHSs. Then, for example, if a monster is wounded, the inference engine can figure out what the monster should do by accessing only those rules whose conditions depend on the monster's health.

FINITE STATE MACHINES. Another useful way to represent a plan is the finite state machine (FSM), which consists of nodes (representing states) and the arcs between them (representing the transitions from one state to another). A node or a state in a FSM can stand for a particular stage in the plan, and an action associated with the node would be the action to be carried out. The transitions occur when an action is completed, or some event interacts with the process of the plan; conditions on those transitions can indicate what the next state should be. Figure 1 shows a simple transition diagram for the same scenario as Listing 1.

EXPLICIT GOAL, PLAN, AND ACTION STRUCTURES. A representation of plans will be more powerful and robust if you can explicitly represent and reason about these aspects of plans, rather than get at them through indirect means. This requires that you represent plans, goals, and actions in some explicit way. There are many ways to do this, and since the needs of different games can

vary widely (even within the same genre), I won't advocate one particular representation.

The following are possible fields that you might include in a structure for a **goal** class:

- A *text string* in which to store the name of the goal, such as "Occupy City." You may need text strings for other needs too, such as comments.
- A *defined constant* representing the goal in a form the program can recognize.
- A set of *slots* that stand for the parameters of the goal, such as the city to occupy in a war strategy game.
- The *bindings* of the slots. A generic **goal** type will have a slot for the city to occupy. A specific instantiation of the goal will state to which city it's referring. The binding could be a pointer, an index to an array, a defined constant, and so on.
- A *pointer to a function* that can determine whether the goal is *satisfied*.
- A *pointer to a function* that *evaluates* how close one is to satisfying the goal, which measures progress. This and the preceding could be the same function, with 100% representing full satisfaction, or they could be different, since progress measurement and satisfaction tests may be more efficiently represented separately.
- A *priority* for the goal, to help decide which goals to work on first.

As mentioned earlier, there are differences between the **goal** class structure, general goal templates, and instantiated goals. The **goal** class structure is defined at compile time within the code. General goals all use this same class but represent different specific goals. Thus, the fields would have different values, such as "defend location" or "attack unit." These goals would probably be created during development and saved in a file, to be read in at run time and allocated as goal templates. Instantiated goals are goals assigned to a specific circumstance. They are like copies of the generic goals with slots bound to specific objects (such as "defend Paris" instead of "defend location"). These two types of goals are probably best represented by having different classes for each, whereby the instantiated **goal** class would have a pointer to the appropriate generic **goal** class as well as the slot bindings. These same principles apply to **action** and **plan** classes as well.

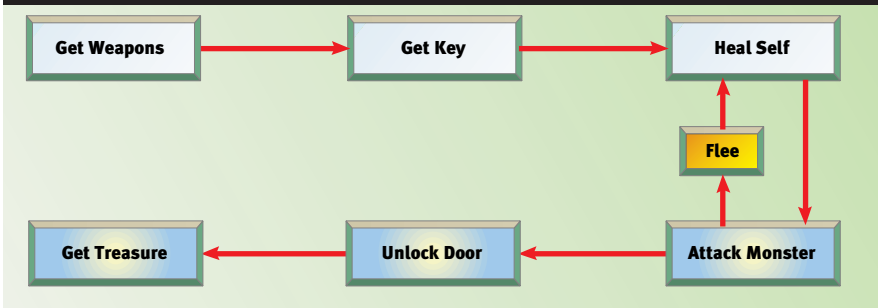
The fields for an **action** class could include:

- The *action name* and other strings.
- A *defined constant* representing the action in a form the program can recognize.
- *Slots* that represent the subject and object(s) of the action, such as the agent doing the attacking, the agent being attacked, allies, and so on.
- *Parameters* for the action. For example, an action to purchase fuel would need to know how much fuel to purchase. The slots and parameters could be represented in the same way.
- *Bindings* for the slots and values for the parameters. These would be for the instantiations, not for the action templates.
- A *pointer to a function* that actually carries out the action (for instance, "monster attacks player").
- The *prerequisites* for the action. If the program is building a plan from scratch, the prerequisites give it additional subgoals for which to plan. (For instance, needing a key to unlock a door makes obtaining the key a subgoal.) While executing a plan, prerequisites provide a test whose failure is a reason to abort the action and perhaps the whole plan (such as the key that is lost or destroyed before you can use it).
- The *changes* the action makes to the game world (for instance, "the chest is now unlocked," or "the city is now occupied"). If the program builds plans, this field is used to look for actions that fulfill goals or preconditions. While executing plans, this field can be used to see if the action's changes have already happened, in which case the action is skipped (for example, there is no need to unlock an unlocked door). The changes should be represented using the same defined constants used for the goals, for easy comparison.
- The *resources* the actions consume, whether raw materials, fuel, money, or something else. The resources could simply be be part of the prerequisites (for instance, "x tons of iron are needed to build the ship"), or they could be represented as a separate type of field. The resources consumed also count among the changes made to the world.
- The *time* it takes to perform the action, if that's important. The time could just be another resource, or it could be considered separately, since time has its own attributes (everyone has the same amount, it can't be traded, and once gone it can't be taken back).
- The *preference* associated with the action. A function that evaluates which action to invoke can judge based on the various attributes, but an extra field can capture information not directly represented, or provide a shortcut past the whole evaluation process.

The fields for a **plan** class could include:

- The *goal* of the plan, such as "capture location x." While a plan is being executed, the goal can be checked so that the plan can be halted if or when the goal is fulfilled, whether deliberately or serendipitously. If a plan is being built, the goal needs an explicit representation, as explained previously, so that the builder can plan for it.
- The *steps* that comprise the plan. Depending on the needs of the game, these may include subgoals as well as low-level actions. It's no coincidence that the fields for goals and actions explained previously have many similar fields. Thus, a class implementation could define **planstep** as a superclass of the **goal** and **action** classes, for use in the plan's **step** field and other places.

FIGURE 1. *Sample FSM for a plan to get past a monster and locked door to a treasure.*



LISTING 2. *Sample plans for a simple dungeon explorer.*

```
Goals:
StayAlive [100]
GetExperience [50]

Plans:
if BadlyWounded do GetHealing to StayAlive [80]
if BadCombatSituation do RunAway to StayAlive [90]
do GetTreasure to GetExperience [30]
do KillMonsters to GetExperience [40]
if IsHealingSource(x) do (GoTo(x), Use(x)) to GetHealing [78]
if IsTreasure(x) do PickUpTreasure(x) to GetTreasure [25]
if IsMonster(x) do AttackMonster(x) to KillMonsters [37]
do ExploreDungeon to GetTreasure [15]
do ExploreDungeon to KillMonsters [15]
if not Explored(x) do GoTo(x) to ExploreDungeon [12]
if DoorClosed(x) do OpenDoor(x) to GoThroughDoor(x) [12]
if DoorLocked(x) do (FindKey(y), UnlockDoor(y,x)) to OpenDoor(x) [10]
```


- The *changes* the plan makes to the world. This is a composite of the changes that individual actions have made which aren't unmade by other actions. Changes made to the world are important to know because actions may have side effects that cause one plan to be chosen over another.
- The *order* in which the steps are taken. The order can be strict (for example, "do A, then B, then C, then D") or it can be a partial order (for example, "do A before D, and B before C"). Partial orders are more powerful and flexible, but they are more complicated to track.
- The *causal links* in the plan's steps. For example, the action `OpenDoor` is fired to fulfill the `Opened(Door)` prerequisite to the `Remove(Chest,Room)` subgoal. These links are useful during plan construction to make sure that nothing affects the action's result before the goal is fulfilled (for example, "make sure nothing shuts the door before the agent can take the chest out of the room").
- *Control flow* information. An advanced planner can include loops and conditional branches such as those found in programming languages (for example, "if the city is found vacated, occupy it; otherwise surround and attack the forces there").
- The *time* and *resources* used by the plan as a whole.
- A *preference* rating.

Of course, not all of these fields need to be used. When you're developing a plan-building or plan-execution system, you're better off starting simply and gradually adding functionality. A simple representation of a plan would be a linear list of steps, assumed to occur in their listed order.

You may have noticed that goals and actions possess a lot of similarities. This is not a coincidence, since either type can be used as a step in a plan. You might want to make both classes descendants of a common ancestor class.

The explicit representations explained previously offer the most direct means of applying the planning algorithms, which I will discuss shortly. The explicit plan structure contains several lists: the steps in the plan (including their order), the causal links, and the variable bindings that refer to the list of steps. The normal trade-offs in data structure construction occur at this point — you must take into account the typical size and range of size of the list, its usage, and so on. Usually, lists are constructed as arrays with their references represented by array indices, or built as linked lists and pointers.

Oh, The Plans You'll Build

Having examined ways of representing plans, let's look at runtime methods for selecting plans, implementing plans, and recovering from failed plans. We won't cover the topic of building plans, even though this was one of the first efforts in planning research. Building a new plan from scratch (that is, from only primitive actions), is time consuming for both AI and for humans. Most plan-making and following that people do is based on the adaptation of previous plans to new situations; similarly, we'll assume that you'll define plans for your game during development and save them to a file for the program to manipulate.

Listing 2 shows the sorts of plans one can build for an autonomous dungeon explorer. The number after the goals and plans show an assigned priority. As the explorer moves about, it acts upon the goals and plans that have the highest priority — thus, if it's in a life-threatening situation, the plans for self preservation will be invoked and followed, but if not, then the lower-priority goals of monster slaying and exploration are pursued.

Note that there are several levels of plans, incorporating different subgoals that can be used. The plans here are very simple, only one or two actions long, but longer plans can be developed. The balance between the number of subgoals, the number of plans, and the length of the plans depends on the needs of the game's intelligence and the efficiency needed for dealing with all the plans. A well-defined set of plans can meet the needs of several genres of games and achieve many AI goals.

- Simple agents in an action game can be given a sense of operating under a set of goals. If their plans are complete enough, they will act in a reasonable way regardless of the situation in which they find themselves.
- In role-playing games, one may assign nonplayer characters a set of goals to work from, such as earning extra money, seeking adventure, getting revenge on enemies, and so on, which makes them seem more like real people with separate lives.
- War games can structure their strategic and tactical thinking around plans. Objectives can be defined and then attacked or defended according to how they achieve some higher goal in the conflict.
- Strategy games can manage resources with plans as well. Plans can manage the economic infrastructure, military buildup, R&D, and so on, according to the player's current goals.
- In both action and strategy games, plans can be used for group movement as well as individual action. For

example, an ambush or a flanking maneuver can be coordinated by using either an overall plan run by a virtual commander who tells the individual agents what to do, or by separate plans that tell each agent what to do when certain other agents have done their parts.

Plan A, B, or C?

Assuming that our game has several plans built, the next issue to deal with is how to choose which plan to follow in a given situation. This issue can be considered from several angles, leaving a wide range of approaches to take.

For instance, should you choose a plan based on optimized or simply satisfied conditions? In other words, should the game look for the best plan of all or a plan that is simply good enough? A satisfying approach would take the first plan which meets some minimal standard — the first to score over a given minimum, or perhaps the first that looks as if it will work. This approach can be helped by examining candidate plans in a particular order, such as from simple to complex, or cheap to costly. For example, in order to capture a city in a war game, the player can first try marching into it (if it's empty of enemy troops), then can try a simple attack followed by an advance, and finally can attempt more involved and prolonged attacks.

An optimizing approach would examine several plans and choose the best one based upon its score. Fortunately, not every possible plan needs to be examined (because the same plan can have many instantiations with different variables or slot bindings, such an optimizing approach can involve huge numbers of plans). The number of plans can be held down further by limiting the search to a certain number of plans or a certain amount of time spent searching for the best plan.

When I refer to a plan's "score," I mean some function that evaluates the goodness of the candidate plan. In short, the score for a plan will consist of its "value" minus its "cost." Value can be measured in terms of the fulfillment of goals, the value and quantity of goods received, the importance of land occupied, the value of good relationships established, and so on. The cost could include resources consumed, time expended, estimated damage taken or lives lost, the number of units committed that could also be used elsewhere, or some other method of measurement. Both the value and cost estimates should factor in an estimate of the certainty that valuable things will be gained or lost if that plan is chosen.

LISTING 3. *Simple algorithm for plan execution.*

```
repeat
        get the next action in the plan
        perform the action until it is finished
until the plan is empty
```

Another issue in the plan selection process is whether to use an agent- or goal-based system. In an agent-based system, an agent tries to decide what to do next, including which goals to work for as well as which plans to follow and which actions to take. The agent is the center of the focus — individual plans or goals may be dropped or modified. In a goal-based system, the focus of attention is the goal, and agents are used in order to achieve the goal, rather than vice-versa. The goal-based system is a more natural approach when there are many small agents at the disposal of an overall deciding entity — it is frequently used in war games and strategy games. For instance, if the virtual general in a war game wants to capture a location, that goal will look for units to use in that effort. The units themselves aren't considered to have individual goals and exist only to follow orders.

Another decision you must make is whether to use a top-down or bottom-up goal selection process. In the top-down approach, a high-level goal may be fulfilled by achieving a few subgoals. These subgoals will have to be broken down into further steps, and so on until actual actions are decided upon. A bottom-up approach works in the reverse direction: It looks at the current situation and decides which actions can be done at the time, sees how these actions might work toward achieving goals or preconditions for other actions, and then builds up plans from there. A combined opportunistic approach is often useful, too, involving either top-down or bottom-up planning as is appropriate. For instance, in a war game, a commander could build a top-down plan to drive through a certain part of the enemy's line. If the enemy responded by bringing in troops from another sector, and thereby leaving that sector too weakly defended, a bottom-up planner should notice the weakness and plan to send a force to puncture the line at that point.

Finally, you must decide whether complete plans, a partial plan, or only specific goals will satisfy your AI. If a situation under consideration is fairly predictable, and the plan takes only a short amount of time, then making a complete plan is useful, and the emphasis is on plan selection. But the more dynamic and unpredictable the environment, the less useful it is to lay out complete plans. In such cases it may be useful to plan in detail only the next few actions and leave other subgoals for later. In extremely dynamic environments, your action selection process is concerned with the very next action. You can plan this in several different ways. In a top-down approach, you choose the best high-level plan for the desired goal, then choose the best subgoal within that plan, and so on recursively until an action is selected. In a bottom-up approach, the action is selected that best fits the current situation and seems to advance toward desirable subgoals and goals. Yet another approach is to look at several possible plans for the current goals and choose the action that fits the greatest number of plans — in other words, the one that leaves the most flexibility for future actions.

LISTING 4. *Algorithm for plan execution with action monitoring.*

```
repeat
    get the next action in the plan
    if the action's purpose or changes have already happened
        continue
    if the action's preconditions are false
        abort the plan
    do
        perform the action
    until the action is finished or time runs out
    if the action's changes are false
        abort the plan
until the plan is empty
```

Acting upon a Plan

Our game AI knows how to represent plans, our agent has chosen a plan that advances toward a goal, and now the agent must act upon that decision. How should this be done? The simple, straightforward approach is shown in Listing 3. The agent repeatedly looks up the next action in the plan, and then does it, until all the actions in the plan are done. This is good enough for linear plans in a static environment.

If the plans aren't linear — that is, if the steps are listed in a partial order rather than in a complete order — then getting the next action is a bit more complex than just reading the next step of the plan. In this case, one needs to look at all the unexecuted actions that have no unexecuted predecessors and choose one of them to do. You could choose the first such action found, or look at several of them and choose the best one according to some game-specific criterion, or choose the one with the highest priority rating (which could be attached to an action or to a production rule if rules are used).

Not all situations are static or predictable, however, and this can affect which plan is chosen. These dynamic situations often crop up in games, where there are usually multiple agents, each with its own agenda. As I stated in my discussion of actions and plans, you can monitor the plan during execution and adjust to violated expectations. The first level of monitoring is action monitoring (an example of which is shown in Listing 4), so called because it runs the tests at the action level and it helps avoid invoking stupid actions such as trying to open a padlocked chest or a chest that is already open. An action monitor performs three types of tests:

1. It tests the action's intended changes. If they're already true, the action is skipped since it would be superfluous.
2. It tests the action's prerequisites. If any of them are false, the action cannot occur, and the plan is aborted. (Note that this may also be put in the search for the next action when using partial ordering: Find an unexecuted action with no unexecuted predecessors, whose prerequisites are true.)
3. It monitors the action's execution, not only to stop the action when the desired changes are true, but also to know when to stop trying. It may quit the action if the preconditions are violated during execution, or if a certain amount of time has passed or certain number of attempts have been tried. (The time-out test can easily be added to the simple loop of Listing 3 without any reference to an action's purpose or preconditions.)

Action monitoring is fairly robust, but it's not perfect. It will detect problems with the current action, but it won't detect problems with future actions. For example, if an AI-controlled general starts assembling forces to take a city, action monitoring won't notice if the enemy abandons the city until the whole force is assembled and is actually at the point of carrying out the assault. For another example, consider an AI-driven character that found a locked chest and went off to look for the key. If the character noticed some imp running off with the chest, the character wouldn't do anything about it until he returned to the scene with a key. Handling such problems requires a more robust form of testing called plan monitoring.

Plan monitors perform tests similar to action monitors, but on a global level. First, they test to see if any upcoming action (or subgoal, if it's not broken down to actions yet) has a violated precondition. This test only applies to preconditions whose causally-linked action has already been executed. If such a precondition is found, the plan has failed. Second, plan monitors test to see if there are steps that can be skipped. A plan monitor checks all unexecuted parts of the plan to see if their desired changes have already occurred. If so, then their preconditions' causal links are followed back and marked as being skipable. When looking for an action to perform afterwards, if all its changes are marked skipable (in other words, every

LISTING 5. *Algorithm for plan execution with plan monitoring.*

```
repeat
    if there is a precondition for an unexecuted action
     whose value is false
     and whose causal action has already been executed
        abort the plan
    from the final goal and working backward
        if the goal's or action's changes are already true
            for each precondition of the goal or action
                mark the action that causes the precondition as skipable
    get the next action in the plan with a needed change
     [ie. with a change not marked skipable]
    do
        perform the action
    until the action is finished or time runs out
    if the action's changes are false
        abort the plan
until the plan is empty
```

plan that might have needed this action no longer needs it), then the action is skipped. If the final goal is found fulfilled, you might want to add a special condition to exit the plan, rather than marking all remaining actions as skipable.

Once a plan is instantiated, it may be a good time to go through and annotate all of the preconditions to be tested before any actions are taken. This avoids having to do it each time an action occurs. Since these tests are more time consuming than action monitoring, you need to test carefully to see if plan monitoring is worth the effort. You may prefer to do only one of these tests in addition to the action monitoring, or to do none at all. If the plan is linear, using a full order rather than a partial order, then it is much easier to do the tests, since it's easy to determine which are the following or preceding actions. You can just go up or down the list, rather than following causal links around. Listing 5 shows how the basic algorithm for plan monitoring might work.

The Best Laid Plans of Mice and Men Go Oft Awry

Or, as Von Moltke put it, "No battle plan ever survives contact with the enemy." However you say it, the truth is that plans often don't work out as they should. Whether from a changing world, an independent agents' actions, or deliberate sabotage, things happen that make plans obsolete. Discovering this condition was the subject of the last section; knowing what to do about it is the focus of this one. There is a variety of different responses an agent can make to recover from foiled plans.

REPLAN. The simplest act one can perform is to completely scrap the old plan and find a new one to instantiate and follow. It's the easiest to program, and in simple situations, it's sufficient. In other situations, though, it may be inefficient — replanning means redoing much of the work that went into the old plan.

However, if one is working with plans in a hierarchical fashion — plans that have have subplans and so on — then the replanning can be fairly efficient. This is because you only have to replan at a low level, leaving the high-level plans intact. If a situation gets really messed up, then plans will fail at higher levels, which may necessitate replanning at the higher levels.

REINSTANTIATE. One of the simplest ways to salvage a broken plan is to find another way to instantiate the variables or slots of the old plan. For example, if you planned to go down a road that is blocked, you might look for a different road. Or, if one agent is too damaged or otherwise involved and cannot attack, then attack with another unit.

FIX THE PROBLEM. Another approach to recovering from a failed plan is to make the violated precondition true again. That is, make the condition a new goal, and find a plan to make it true. For instance, if the road you wanted to travel down is blocked, find some way to remove the barrier.

TRY ALTERNATIVE PLANS. You may switch from one plan to another. For instance, if the dungeon crawler's key doesn't unlock the door, she may set off in search for another key, or she may decide to try destroying the door with her axe!

USE CONDITIONAL PLANS. The original plan can have conditional tests meant to handle contingencies, thereby avoiding the cost of determining what to do when a problem arises. For instance, the dungeon crawler may decide to bring along all her keys, her axe, and a gunpowder bomb to deal with possible problems. The drawbacks to this approach are that you can't anticipate all possible problems, you can't prepare for all the problems that you can anticipate, and anticipating many problems simply may not be worth the time to think about them or the resources to deal with them. When deciding which problems to anticipate, it might be useful to consider both their probability and their impact. For instance, if a party of dungeon crawlers splits up, it might be important for each person to carry a weapon, not because they're likely to run into a monster, but because it would be fatal to encounter a monster while unarmed.

ABANDON OR POSTPONE THE GOAL. This is a simple solution to the problem — give up and do something else. It may be that circumstances within the game will change and the goal will become achievable later.

All of these approaches need not be mutually exclusive. A routine for dealing with plan problems may consider several of them — reinstantiation, fixing the problem, using alternative plans, and dropping the goal — and choose the option with the best cost/benefit trade-off.

This coverage of AI planning should give you several ideas to play around with — some of them should be applicable to your particular game situations. Good luck! ■

FURTHER READING

This overview has been necessarily general. Here are some books which are very useful for further reading:

Allen, James, James Hendler and Austin Tate. ***Readings in Planning.*** **(Morgan Kaufmann, 1990.)** A very good collection of important historical papers covering various aspects of planning.

Shapiro, Stuart C. ***Encyclopedia of Artificial Intelligence.*** **(Wiley Inter-Science, 1990.)** Good article on planning (and many other good articles).

Russell, Stuart and Peter Norvig. ***Artificial Intelligence: A Modern Approach.*** **(Prentice Hall, 1995.)** This has the best discussion on planning of current AI texts. In fact, I regard it as the best of all current AI texts, period, both in general and for computer game developers. It has the widest selection of topics, the most recent research, and uses *agents* as its organizing paradigm – rather than presenting the different fields of AI in a loosely-related manner, it treats all of them from the viewpoint of software entities that are trying to perceive their environment, make decisions, and act — just the things that games are full of.

Avoiding DirectSound3D Database Traps

by Rich Warwick

3D audio can have a tremendous effect on a gamer's experience. Unfortunately, if a game doesn't use the DirectSound3D API correctly, the effect can vary between little or no 3D audio positioning to even more serious problems, such as accidental system overloading. While the methods for implementing 3D positional audio via DirectSound3D can be found in various books as well as in Microsoft's own documentation, knowing how not to write to this API is just as critical to a successful implementation in your game.

Defining the Vocabulary

Before diving into how and how not to use 3D audio, let's clear up some of the confusion associated with the technology. Often the terms "3D audio," "positional audio," "spatialization," "virtualization," and "stereo expansion" are used interchagably. In reality, 3D positional audio, virtualization, and spatialization are three different concepts, and their differences must be understood before they can be properly applied to games or applications.

Spatialization, sometimes called stereo expansion, uses signal processing to expand the perceived location of speakers. It is a nonlocalizing effect, meaning that it doesn't localize a sound or a channel to a specific location. In fact, it does the opposite. Spatialization disperses the perceived location of the sound so that the listener can no longer determine the exact location of the speakers. It makes the listener believe that the sound is coming from an area that is much wider than the actual speakers. The general perception of spatialization is that it makes a stereo stream sound much richer to the average listener.

Virtualization uses signal processing to fool the listener into thinking that there are other speakers present that aren't really there. For example, a system could use only two front speakers or headphones and virtualize rear or surround sound speakers for a home theater effect. Virtualization localizes a specific channel of audio (such as left rear) to a specific location, as opposed to localizing a specific sound to an exact location (which is what occurs in 3D positional audio). Virtualization is only used when the source media has more than two channels of audio — its usefulness for positioning interactive sound sources (such as those found in action games) is limited. Examples of multichannel source media are Dolby Prologic

Rich Warwick's experience in the PC industry spans chip, gate array, and board level hardware design, as well as DSP software development. Rich joined Crystal Semiconductor Corporation in 1995 as a software development manager. He was responsible for the software development for Crystal's CS4610 PCI audio accelerator. He is currently the Manager of DSP Software Technology, leading the development of software for Cirrus Logic's "Sound Fusion" line of PCI audio accelerators.

Surround or Dolby Digital audio streams. In this article, when I refer to 3D audio, I mean 3D positional audio.

3D positional audio uses signal processing to localize a single sound to a specific location in three-dimensional space around the listener. 3D positional audio is the most common effect used in interactive games, because a sound effect, such as the sound of an opponent's automobile, can be localized to a specific position. This position, for instance, could be behind the listener and quickly moving around the left side while all the other sounds are positioned separately.

3D positional audio is also referred to as HRTF-based 3D audio. HRTF stands for "head-related transfer function," a method by which sounds are processed to localize them in space around the player. Although this technique is acceptable for 3D positioning, it requires a large amount of processing power. This is the reason 3D audio hardware accelerators are becoming so common in PCs. For an explanation of the mechanics of 3D audio and HRTF processing, see "Exploiting Surround Sound using DirectSound3D" in the December/ January 1997 issue of *Game Developer*.

APPLICATIONS FOR SPATIALIZATION. Spatialization is an effect for processing music, such as an audio CD or a stereo music soundtrack in a game. This effect is especially useful for PC speakers that are built into the monitor because it makes the sound appear to come from a much wider sound field than the actual speakers, which in this case are very close together.

Care must be taken never to apply this effect to an audio stream that has already been processed by a 3D positional algorithm, because spatialization alters the phase of the signal and can destroy the 3D positional effect. However, such conflicts are the concern of the audio system designers, not the developer of game or application.

APPLICATIONS FOR VIRTUALIZATION. Virtualization is an effect that gives the listener the impression of a home theater environment even when only two speakers or headphones are present, which is typically the case with multimedia PCs. However, to use this effect, multichannel audio must be available, and the sounds to be played back on the virtualized rear speakers must be encoded onto those tracks during production. This makes this solution less than ideal for the action portion of games, in which sounds might have to jump from the front speakers to the rear (depending on the player's actions), but cannot due to prior encoding on a specific channel. However, multichannel audio and the virtualization of these channels are very effective for noninteractive game intro scenes.

Virtualization is typically used to play back Dolby AC-3 or Dolby Pro Logic audio streams on a system that only has two speakers. For example, DVD movies can have a 5.1 channel Dolby AC-3 audio track encoded on the DVD. (The 5.1 channels are actually left front, right front, center, left rear, right rear, and a subwoofer. The subwoofer is referred to as the ".1" channel of the 5.1.) 3D positional audio techniques are used to position a front center channel as well as right and left rear channels at their virtual locations. Virtualization simulates the additional speakers that aren't typically present on a computer. Virtualizing rear speakers is an example of using 3D positional audio for a noninteractive application, because the virtual speaker locations aren't moving or responding to the listener.

APPLICATIONS FOR 3D POSITIONAL AUDIO. One of the reasons that 3D positional audio is so popular in action games is because it can be interactive. Sounds don't have to be preprocessed during the game's development to position the sound. As the listener changes location in a virtual world, all the sound objects can maintain their correct location speed and path of motion around the listener as the action unfolds.

This is different from applications that encode the audio into a certain channel (such as a rear or surround channel) during development. Encoding the location of a sound into a particular channel of a multichannel stream is an example of noninteractive audio placement. Multichannel audio is typically used in an environment where the listener doesn't have control over the sounds — such as when you watch a movie in a home theater.

Implementing 3D Audio Today

The use of 3D audio in PC games initially wasn't widespread, mostly due to poor 3D audio support in the Microsoft DirectSound3D 3.0 API. The first iteration of this API didn't allow specialized 3D audio hardware to process the 3D streams, and as a result, game developers couldn't be certain that 3D sound objects would sound satisfactory and have the desired effect — even if a world-class 3D audio accelerator was present in the PC. So there wasn't much incentive for game developers to incorporate 3D audio into their games early last year. However, when DirectSound3D 5 started shipping in August 1997, the situation changed. That API supports specialized 3D audio accelerators for processing 3D audio streams.

Using the DirectSound3D API

While there are a number of sources for information about using the DirectSound3D API, it's equally important to know how not to use it. At last year's Computer Game Developers' Conference, a number of programmers working on 3D positional audio for games explained problems that they had encountered and the lessons that they had learned during development. Their problems resulted in poor performance in their games and little or no apparent 3D effect, even on the sounds the developers *most* wanted to position in 3D space. I've boiled their comments down to four rules that you should follow when implementing 3D positional audio in your own title.

1. Never use variable frequency buffers unless they're actually required. It's important not to request variable frequency sound buffers when they're not necessary. This is a common mistake because it's the default in some of the DirectSound SDK examples. To make best use of a hardware DirectSound accelerator, pay attention to the flags specified in the `lpcDSBufferDesc.dwFlags` parameter, which are passed to `IDirectSound::CreateSoundBuffer()`. Some of these flags are straightforward — for example, `DSBCAPS_LOCHARDWARE` asks for a hardware-accelerated buffer and `DSBCAPS_CTRL3D` asks for DirectSound3D control. Other flags have implications for hardware accelerators that aren't so straightforward. The worst offender is `DSBCAPS_CTRLFREQUENCY`, which specifies that you need to be able to adjust the buffer's playback frequency after the buffer has been allocated. There are circumstances where this is a useful capability (some racing games use it to adjust the pitch of an engine, for example), but most of the time it's not required.

You also need to be aware that Microsoft's default `lpcDSBufferDesc.dwFlags` value, `DSBCAPS_CTRLDEFAULT`, includes the `DSBCAPS_CTRLFREQUENCY` flag (in addition to `DSBCAPS_CTRLVOLUME` and `DSBCAPS_CTRLPAN`), so you're not safe from this potential performance-draining trap if you supply Microsoft's default flag value.

Every buffer that is created with `DSBCAPS_CTRLFREQUENCY` (that is, set for variable frequency) will assign a sample rate conversion application to the audio stream. This consumes additional processing resources either on the host processor or on the audio hardware accelerator. If the host PC contains a multitasking audio hardware accelerator with dynamic resource management, the resources consumed by these sample rate conversion applications could have been used to accelerate more 3D audio channels. As you can see, blindly creating all streams as variable frequency streams will slow down the host processor and/or prevent 3D audio streams from being accelerated.

2. Never use 3D when 2D will suffice. Only create 3D sound buffers when necessary. Many games have created all of their audio streams as 3D streams, but this practice results in a poor listening experience. An accelerated system will quickly run out of resources, and the result will be a complete lack of accelerated 3D audio. After all the hardware accelerator resources are consumed, the host will apply its 3D audio algorithm to the remaining streams. These streams will consume much more processing power than 2D streams, and further burden the system. The bottom line is that it's very wasteful to play every stream as 3D when it isn't necessary.

TABLE 1. *Games that currently support 3D positional audio or will do so in coming months.*

Company	Game
Acclaim	Turok: Dinosaur Hunter; Forsaken
Activision	Heavy Gear
Crack dot Com	Golgotha
Eidos	The Dark Project
Electronic Arts	Moto Racer
GT Interactive	TigerShark; MageSlayer; Bug Riders
Interactive Magic Online	WarBirds
Interplay	Starfleet Academy
LucasArts	Outlaws, Jedi Knight
Maxis	SimCopter; Streets of SimCity; SimCity 3000
Microprose	MechWarrior III
n-Space	TigerShark; Bug Riders
Probe	Forsaken
Raven Software	MageSlayer
Reality Bytes	Dark Vengeance
Ripcord Games	Space Bunnies Must Die
Sculptured Software	Turok: Dinosaur Hunter
SegaSoft	Rocket Jockey
Sony Interactive Studios	Tanarus
Techland Software	Crime Cities; Speed Thrill; Virtua Command; Crusher; Speedway Manager 3D

Source: http://www.aureal.com/tech/A3Ddevs.html

3. Avoid using the DuplicateSoundBuffer call. A common mistake that game developers make is not checking for the failure of a **DuplicateSoundBuffer** call. Failure to check for a bad return code from this call can lead to audio streams or sound effects being completely lost. The reason an application developer would use the **DuplicateSoundBuffer** call instead of using **CreateSoundBuffer** is strictly a matter of convenience. Instead of providing all the necessary parameters for each 3D buffer, you can just reference, or duplicate the parameters of a previously created buffer. This practice is acceptable only if you understand what happens when a call fails and can take corrective action.

Here's the problem: If a new 3D sound buffer is created using **CreateSoundBuffer**, the buffer will be created on the hardware accelerator if there is a free 3D channel. If no hardware 3D channels are available, the software 3D emulation in DirectSound 5 will automatically take over. However, if the new 3D sound buffer is created using **DuplicateSoundBuffer**, the host emulation in DirectSound 5 cannot take over. This is because DirectSound 5 doesn't have access to the parameters that you're requesting to duplicate. These parameters only reside in the hardware accelerator. Therefore, the **DuplicateSoundBuffer** will fail, and DirectSound 5 won't play the buffer on the host. At this point, the buffer is lost and the sound will never be heard. This isn't a problem if the game or application checks for the failure and then reinitiates the call using **CreateSoundBuffer** instead. However, the best approach is not to use the **DuplicateSoundBuffer** call. Only use **CreateSoundBuffer**, and this problem can be avoided.

4. Always create the most important 3D sounds first. To maximize the 3D impact on the listener, it's important to create the most important sound buffers first. Each time a 3D sound buffer is created using **CreateSoundBuffer**, the buffer will be created on the 3D audio hardware accelerator — if one is present in the system and has a 3D channel available. If all the hardware accelerated channels are already consumed, or there was no accelerator in the system, the host 3D algorithm in DirectSound 5 takes over and processes the stream on the host. This process is invisible to the game or application. The steams that are relegated to the host won't be positioned very well and will use more host processing power than 2D streams.

Follow the Rules

Always create the most critical 3D sound buffers first. That way, if there is an accelerator in the system, it will be used for the sounds that will have the most impact on the listener.

It's no secret that 3D audio can be an immersive experience for game players. However, if you're going to adopt this technology, make sure that you avoid the common pitfalls associated with DirectSound3D. When followed, these four rules will maximize the impact of your audio and reduce unintended audio processing. ■

A Chao Theo

by Martin Streicher

Listen to this and see if it sounds familiar. You want to create a new game. You think your game is going to be a hit, the next big thing. You've assembled a team of talented people, who have lots of great ideas and have created some fancy technology. All that you need is time and money. You present your idea. It's well-received.

And, you can have your money if... you can ship your game by Christmas 1998. You glance at your notes and your impressive-looking Gantt charts, and then confidently say, "No problem." If only that were true.

If you're developing a new, innovative game, you're going to have problems. You'll make mistakes. You'll struggle to find solutions. You'll design and redesign through the process of trial and error. You'll be forced to make trade-offs. Your plans will change. The unexpected will happen. You'll be faced with crisis. You'll have to divert at least one disaster.

When your goals are ambitious, chaos and crisis are the norm, and anxiety is a constant companion. Creating a new game can be nerve-wracking, even perilous (financially, physically, and emotionally). But taking risks also provides you and your development team with challenge, excitement, and enrichment — perhaps even fame and fortune.

In my September 1997 *Game Developer* article, "The Game of Risk," I presented some techniques to identify and manage risk in your development projects. In this article, I present some techniques for encouraging, embracing, and leveraging risk and chaos in product development.

Martin Streicher is an Executive Producer at Berkeley Systems, Inc. in Berkeley, CA. He graduated from Purdue University in 1986 with a Master's degree in Computer Science, and has been a software development manager since 1989. Most recently Martin produced YOU DON'T KNOW JACK MOVIES and YOU DON'T KNOW JACK VOLUME 3. Martin is currently the executive producer, producer, and director of a new CD-ROM game show that he also created. Please send comments or questions about this article to strike@berksys.com.

Misery Loves Game Companies

Game development schedules are notoriously volatile. Why? Because creating great games is an art. And while that comparison may be overused and cliche, I think it's apropos.

The people that create games — game designers, developers, artists, animators, modelers, musicians, and writers — are all artisans who have mastered a specialized craft. Like other craftspeople, the people that create games require skill, insight, direction, materials, and time to produce great work. I would argue that game developers face even greater challenges since their materials — hardware, software, storytelling, game play, art — are so diverse and often revolutionary. A new game makes the unreal real — certainly the stuff of art.

Additionally, the process of creating a game — its design, execution, and implementation — is not a science. Each software development "model" has advantages and disadvantages, and development methodologies vary greatly. Great games challenge the limits of technology — you should expect that creating great games will likewise challenge your status quo and the limits of your own abilities. If your goals are ambitious, then you should expect things to change. You should expect to adapt your current practices and invent new ones.

You'll certainly face situations and problems that you've never faced before, and you'll be called upon to make decisions for which you're unprepared. You'll make mistakes, but that's normal. Game development has no formulas to follow, nor physical laws to obey. Development has no rules. There are no good nor bad decisions per se — it's up to you to decide what works and what does not work.

All of this uncertainty can be terrifying, but it can also be compelling. Each open issue provides you with the opportunity to excel, create, invent, and imagine. If you're a smart development manager, you'll recognize the chaos, relate to the anxiety it can cause, and at the same time, harness its energy. Trial and error is good chaos. It shows that you're seeing faults in your design and trying to address them. Tweaking performance is good chaos. Building several game prototypes is good chaos. Developing more than one backstory is good chaos. Juggling priorities is good chaos. Finding additional funding to continue development is great chaos. Even if you try and fail several times, your final solution ultimately improves your game.

So how do you do encourage risk-taking, and embrace chaos and uncertainty without derailing your development project? Good question.

To leverage chaos during a project, you must budget it and then manage it — just as you budget and manage money.

Budgeting Chaos

How do you budget chaos? You establish limits for chaos and then decide how much chaos each part of your project can afford. Tasks that you want to control closely are "allocated" little or no chaos. Tasks that require trial and error to perfect are usually afforded more chaos.

For example, you may choose to "allocate" a good portion of your chaos budget on a task such as level design. However, a critical task such as 3D engine development might be afforded less chaos since it's critical to your title's game play and look-and-feel. To create a chaos budget, you must first

PEZ

Things to accomplish:
- Ship PEZ by Christmas 1998 (sound familiar?)
- Stay within budget
- Develop a strong team of writers
- Create a supportive and collaborative culture for content development
- Develop techniques for content testing and refinement

Critical dates:
- 6 October - PEZ funding presentation
- 1 December - End of feature testing; finish all refinements and complete game play specification
- 7 January 1998 - End preproduction phase; start production phase
- July, August 1998 - External prereleases
- September 1998 - Product ships

People required:
- 3 full-time artists/animators/modelers
- 1 part-time artist
- 3 software developers, a production coordinator
- Myself (acting as producer and director)
- A Software Quality Assurance (SQA) lead and several SQA engineers
- A composer and musicians
- A sound engineer
- An editor-in-chief
- 2 full-time writers
- Several part-time writers

Equipment required:
- A ¾" tape deck
- A video capture board
- A dedicated image scanning system
- A recording booth
- A Web-based database server

Project risks:
- Developing a writing staff and creating content
- Developing and implementing an effective content test plan
- Time (time is always a risk; and if you are independent game developer and not a developer/publisher, money is also always a risk)
- Creation of the marketing plan

identify everything that's critical to the success of your project. Specifying what is critical sets limits and ensures that you don't spend so foolishly that it endangers your project.

At a minimum, your list of critical items should include:

- A thorough and detailed description of the new product's features
- A list of critical dates (this list includes your ship date and other milestones such as "feature complete," "external prerelease," and other)
- A definitive list of things that you want to accomplish in the project
- A list of risks to the project
- A list of dedicated resources that are required to execute the project plan (resources include money, people, information, and equipment)
- A list of the expectations that you have for each person on the development team

For example, the sidebar shows the critical items that I identified for a development project that my team just started (the product is code-named "PEZ").

As you can see, my list of crucial items includes the goals of the project, a description of the product that I want to build, critical dates in the schedule, and the resources required to be successful. Armed with this list, I'm happy to let chaos reign free (or in my analogy, be spent freely) until it conflicts with or threatens anything crucial to the project.

My lists are admittedly very broad in scope; they define limits, but are too general for day-to-day chaos management. To complete a chaos budget, your team's lead programmer, lead artist, and marketing manager should compile their own lists of crucial elements. When all of your lists are combined, the entire team can operate independently within well-defined boundaries. In fact, reviewing all of the critical items and individual "chaos budgets" is an excellent way to manage the entire project.

As project requirements change — and we know that they will — redefine your critical items and reallocate chaos as if you were shifting money between investments. Audit how people are spending their chaos. Work together to manage chaos as if it were money.

Consider an extreme example (but not necessarily an uncommon one): You are developing a new game scheduled for a Christmas 1998 release. Shortly after you begin development, you learn that your company is running out of cash and that your ship date for the game is being moved up to Summer 1998. Given this new information, you and your team have to review and revise all of your project goals and your chaos budgets. You may decide to cut features, expand the staff, and accelerate development of certain key technologies. You may decide to take more risk (spend more chaos) in engineering to match the accelerated scheduled. Or, you may decide to severely cut your chaos budget by greatly restricting the kinds of decisions each part of the team can make independently.

Budgeting chaos is oddly similar to how the Federal Reserve Board controls interest rates. If risk is costing the team too much time, effort, and money, cut back on how much chaos is available. If your project needs or wants more experimentation, then make chaos more readily available.

You may be wondering why my detailed PEZ development schedule wasn't included in my list of project parameters. In a chaotic environment, I don't necessarily care when or how things happen as long as the team is achieving its stated goals. Am I advocating anarchy? No. Process and infrastructure are required for any large-scale project. You still need design documents, schedules, reporting structures, and milestones. Ultimately, however, a development schedule is only an estimate of how and when work will be done. A pro forma schedule (such as those usually created in Microsoft Project) is useful only because it initiates a process in which you and your team analyze, estimate, debate, design, review, and ratify a development plan. Once your team has completed the process of creating a project schedule, the schedule is useless.

A development schedule, no matter how detailed, is not a substitute for your leadership. All of your infrastructure — Gantt charts, spreadsheets, status reports, e-mail, code reviews, design reviews, staffing, and meetings — are only a means to an end.

Managing Chaos

To manage chaos, you have to earn it, invest in it, keep track of your investment, invest more when it's needed, and know when it's time to cut your losses. And, to continue the analogy, you cannot manage your investment effectively and safely without doing your research. As the project leader, it is your primary responsibility to measure progress, assess risk, prevent and anticipate problems, and adapt the project whenever a critical item is threatened. It's your responsibility to lead, participate, communicate with, and listen to your team. In fact, accommodating chaos and encouraging risk-taking puts an even greater onus on you to interact with everyone on your development team. Here's how I recommend you manage your project portfolio.

ENGAGE AND PARTICIPATE. As the project leader you must establish and then protect the autonomy and independence of your development team. To establish autonomy, you must establish a creative, collaborative environment that fosters innovation, risk-taking, and experimentation. Enable all of your team members to make as many independent decisions as possible. Once you have autonomy, do not treat it as an entitlement. You have to work as a team to protect it. For example, during the development of YOU DON'T KNOW JACK MOVIES, the art team found itself five calendar days behind schedule. Rather than ask for additional personnel or time to complete the work, the art team decided to work two weekends in a row to make up the time. The art team devised its own solution, exercised its autonomy, and simultaneously increased the entire team's cachet of credibility. The more credible your

team is, the more autonomous it will become.

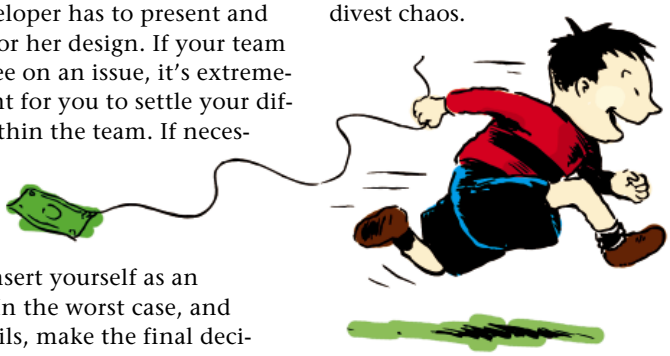
Work with your development team first to find solutions to problems, and try to solve your problems without asking for additional resources (money, people, time). However, if you do need help, by all means ask for it. Asking for assistance doesn't discredit your team — in fact a team that asks for help when it needs it actually reinforces its credibility.

If you've established a collaborative, independent environment, your team should be able to disagree, debate, and reach consensus on its own. I encourage you to foster debates to find the best possible solutions. Let your software developers argue about implementations — you'll get better results if each developer has to present and defend his or her design. If your team cannot agree on an issue, it's extremely important for you to settle your differences within the team. If necessary, study the facts, gather opinions, and then insert yourself as an arbitrator. In the worst case, and if all else fails, make the final decision. Settle the conflict and rally the team to support the decision.

COMMUNICATE. As the project leader, it's your responsibility to defend your development team's priorities, opinions, and decisions. And to best represent your team, you must be intimately familiar with every aspect of the project. Personal contact and interaction with every member of your team is the most valuable technique for gathering and disseminating information. Without information, no team member, including you, can make effective and appropriate decisions. Make sure to communicate regularly with every team member.

For example, I prefer short, one-on-one ad hoc discussions instead of large meetings. I use these quick, frequent meetings to ask and answer questions, gauge status, offer advice, change priorities, and assess risk. Talking to everyone on the team is time consuming, but it's extremely easy to do. For me, it's one of the most satisfying parts of my job. I socialize, learn, discuss, and teach every day.

COORDINATE. Regular communication with everyone on the team also allows me to coordinate the team. When you are managing chaos, this is perhaps the most important task to perform. If you don't communicate and coordinate, you won't be able to measure just how chaotic your project really is. As the project leader, you have to simultaneously envision the "big picture" and scrutinize the finest level of detail. Do you remember my critical items for PEZ development? That's my big picture. The lists that the leads created are their big pictures, yet provide me with another level of detail. By talking to everyone on the team, I assemble both a panoramic and microscopic view of the project. And that's how I manage chaos in my projects — I always know where to invest or divest chaos.



REVIEW. Ultimately, your job as the project leader is to control chaos and ensure that the goals of the project are being accomplished. You need to spend a good deal of time collecting information, and once you have it, you need to analyze it and react. Review and answer these questions every day: What worked today? How can that be leveraged? What didn't work? How can we better solve or prevent the problem in the future? Is each team member focused on his or her critical list? Did the parameters of the project change?

Reining in Chaos

Regular communication with everyone on your team provides you with qualitative results — information such as status, plans, design decisions, and risks. To get quantitative results, you have to analyze your progress against its stated goals. The best way to rein in chaos and yield an accurate appraisal of your progress is to try to build a running, playable version of your game.

Set a deadline, broadcast it to your team, and then focus the entire team on integrating all completed work. Take your art and process it with your tools. Take the data from that step and drop it into your graphics engine. Attach AI to the characters in the game and enable the user interface. Take the output from your level editor and drop that into your engine, too. Start the game and see what happens. Some things will work, others won't. Look at the game closely. Audit your team's progress. Has the game improved since the last incremental build? Why? Why not? Did the team make the right design decisions? Is anything missing? What adjustments need to be made? In staff? In assignments? In priorities? Is the project too chaotic? Incremental builds are excellent indicators of how your project is proceeding. Incremental builds as are like mile markers along a highway — each incremental build gives you an indication of your project's speed and location.

There are no rules for game development, no physical laws to obey. The excitement and challenge of game development is to solve novel, hard, and critical problems with limited time, money, and people. At the start of your project, identify what is crucial to your success. If you remain true to your original intentions and manage with your end-result in mind, how you ultimately accomplish your goals is arbitrary.

Manage time, people, and tasks with your goals in mind. Otherwise, let chaos reign. ■

FURTHER READING

I highly recommend the following books if you are interested in improving your ability to create and manage a chaotic environment.

Maguire, Steve. Debugging the Development Process. Redmond, WA: Microsoft Press, 1994.

McCarthy, Jim. Dynamics of Software Development. Redmond, WA: Microsoft Press, 1995.

Peters, Tom. The Tom Peters Seminar: Crazy Times Call for Crazy Organizations. New York, NY: Vintage Books, 1994.

Yourdon, Edward. Rise and Resurrection of the American Programmer. Upper Saddle River, NJ: Prentice Hall, 1996.

STAR WARS: SHADOWS OF THE EMPIRE

by Mark Haigh-Hutchinson

Shadows of the Empire is an action game originally developed for the Nintendo 64 video game console. It formed part of a multimedia Star Wars event consisting of a novel, soundtrack, toy line, comic books, trading cards, and other related merchandising. The Nintendo 64 version was released in December of 1996, and has proven to be very popular with over one million copies shipped to date. The IBM PC version was released in early September of 1997, and has enhanced cut scenes, Red Book audio (both music and voice), and high-resolution graphics. It requires the use of a 3D accelerator card.

Dinner in Kyoto, Japan, August 1996. (Left to right: Don James, Hiro Yamada, Mark Haigh-Hutchinson, Shigeru Miyamoto, Kenji Miki.)

Why Shadows?

Back in the summer of 1994, LucasArts was exploring the possibility of developing a new 3D title for one of the emerging "next-generation" platforms. After some discussion, the Nintendo 64 was decided upon as the platform of choice, even though there was no hardware available at the time. Due to our close relationship with Lucasfilm, we were aware that Lucasfilm Licensing was planning the Shadows of the Empire event. Jon Knoles, the lead artist and designer on the Nintendo 64 game, took an active part in deciding the timeline of Shadows. He suggested that it take place between *The Empire Strikes Back* and *Return of the Jedi*.

The Shadows story line deals mainly with the criminal underworld of the Galaxy, and the new period allowed us to explore some of the things that weren't explained in *Return of the Jedi*. It also opened up some new characters that were not bound to the original story, which gave us more creative freedom than using established figures. A bonus was that it allowed us to make use of everyone's favorite bounty hunter, Boba Fett.

Since we were developing one of the premier titles for an entirely new game machine, there was a conscious decision to attempt to stretch out and cover a number of different game-play styles. We wanted to ensure that the player would have as much variety as possible, yet still enjoy a satisfying experience.

A Reality Engine for $200?

By early September 1994, we had received our Silicon Graphics workstations and the core team was working. Initially the three programmers were using Indigo 2 Extremes, with 200mhz CPUs, 64MB of RAM, and 24-bit graphics. Eventually, we would have to change our programmers' computers to INDYs (still powerful machines) to install the Nintendo 64 development systems.

In addition, we were fortunate that LucasArts allowed us to obtain a Silicon Graphics ONYX supercomputer. This impressive and somewhat expensive refrigerator-sized computer boasted Reality Engine 2 graphics hardware, four R4000 CPUs, and 256MB of RAM. It became an essential part of our development equipment, as it was the only hardware available that could possibly emulate how the final Nintendo 64 hardware would perform. Indeed, Nintendo and SGI supplied us with software that emulated most of the features that the real hardware would support.

In late September, the programmers took a trip down to Silicon Graphics to discuss the Nintendo 64 hardware design with its chief architect, Tim Van Hook. The SGI engineers were rightly proud of their design, and promised that they would deliver hardware matching the ambitious specifications. Nine months later, we learned that they had indeed met those specifications.

By Christmas of 1994, we had the basis of the first level of the game, The Battle of Hoth, running quite nicely on the ONYX — "quite nicely" being in high resolution (1280×1024), 32-bit color, and at 60 frames a second. By this point, we had also received a very early prototype of the Nintendo 64 controller. This consisted of a modified Super Nintendo controller with a primitive analogue joystick and Z trigger. Due to our strict nondisclosure agreement, we were unable to discuss the hardware or the project with anyone outside the core team. Consequently, we would furtively hide the prototype controller in a cardboard box while we used it. In answer to the inevitable questions about what we were doing, we replied jokingly that it was a new type of controller — a bowl of liquid that absorbed your thoughts through your fingertips. Of course, you had to think in Japanese....

In July of 1995, we received our first actual hardware as a plug-in board for the INDY. This later became known as the Revision 1 board, but on inspection it was extremely "clean" — no wire wraps or other temporary items in sight. Within three days, technical lead Eric Johnston and second programmer Mark Blattel had ported the game to the actual hardware. It was an awe-inspiring moment when we first saw the Battle of Hoth running on the "real" machine. The first revision of the hardware was very close to the original specifications supplied by SGI. Other than the RCP (Reality CoProcessor) not running at quite the final speed, and one of the special video "dither modes" not being available, it performed extremely well.

Over the next few weeks, we would receive an additional two boards, so that all the programmers were developing in a similar fashion. Three months later, we would receive Revision 2 boards, which brought the RCP up to full speed as well as fixing a few minor bugs. Another pleasant surprise was the doubling of the amount of RAM to 4MB.

A further development was the hardware "dither modes" that perform several different kinds of functions at the video back end — mostly to reduce the effect of Mach banding, which is common when using 16-bit color.

Technology

Since Eric Johnston and Mark Blattel had extensive experience with the SGI platform, we undertook to prototype the game using the Performer 3D API. This is an OpenGL-based system that is very flexible. Eventually, we would write our own

Mark Haigh-Hutchinson is a Project Leader and Senior Programmer at LucasArts Entertainment Company. He has been developing computer and video games as a hobby since 1979, and professionally since 1984. Cutting his teeth on numerous 8-bit computers such as the Sinclair ZX Spectrum, he has contributed to 32 published games, 16 as sole programmer. He may be reached at mhh@lucasarts.com.

Jon Knoles directs actor Amos Glick who recoils from an imaginary shot. Mark Haigh-Hutchinson wrangles the cables and provides a supporting hand.

subset of Performer's functionality on the Nintendo 64. This allowed us to move the game from a $140,000 SGI ONYX to a $200 Nintendo 64 in a matter of just three days.

Level designers used the tool set from DARK FORCES to construct the first-person levels for the game. This allowed a crude form of preview using the actual DARK FORCES engine on an IBM PC. This worked fairly well, although later in the project we were able to have a single SGI for dedicated use by the level designers. The PC solution, however, was also useful because the level designers were already familiar with the processes involved. Unfortunately, since the game engine wasn't running on the PC at that point, the development cycle was somewhat slow.

Additionally, the ONYX calculated the preculling visibility tree for each of these levels. The way it works is quite elegant, thanks to Eric and Mark. The world is subdivided into "sectors" — that is, polygonal regions defined by either geometry or some other criteria. These sectors control collision detection, have properties relating to game play, and perform several other related functions. The visibility program traverses the world rendering the scene from the center of every sector in a 360-degree arc as well as three elevations. For every polygon to be rendered in the scene from a particular sector, an identifying 32-bit value, rather than texture information, fills the appropriate pixels in the frame buffer. It's then a simple matter of reading the frame buffer to determine which sectors are visible from that location. This process became known as "pastelization" because the identifiers written into the frame buffer (effectively as RGBA values) caused the scene to appear as purely pastel colors.

Motion Capture

In the spring of 1995, we decided to experiment with the use of motion capture to control the animations of the main character as well as enemies such as Stormtroopers. Fortunately for us, our sister company, Industrial Light & Magic (ILM), had a capture system available for use. It was a tethered system, using a magnetic field to determine the position of each of the sensors. The sensors were attached to the actor at 11 locations using a combination of a climbing harness, sports joint supports, bandages, and Velcro strips.

The nature of the system presented several problems. First, the actor had to perform on a raised wooden platform, since the metal construction supports in the concrete floor would affect the capture system. Secondly, since the actor was on a platform as well as tethered, we couldn't obtain a "clean" run cycle. Some of our more ambitious motions also proved problematic. On the positive side, once the system was calibrated, we were able to capture over 100 motions in a single day, each with two or three different "takes." We viewed the motions in real time on a SGI Indigo 2 Extreme computer running Alias PowerAnimator. This allowed us to quickly ensure that every capture was "clean" before continuing with the next action.

Unfortunately, we were to discover that after analysis, the motion data proved to be unusable. This was mainly because the angle information for the joints wasn't consistent on its representation of the direction around each axis. Consequently, all the animation for the characters was redone by hand, a somewhat time-consuming task.

MIDI Music

Our initial approach to music for the game was similar to that taken on some of our PC titles — namely, a MIDI-based solution. However, the first problem that we came across was hardware incompatibilities between the MIDI keyboards used by our musicians and the Silicon Graphics computers used to develop the game. The theory was that the compositions could be previewed directly on the Nintendo 64 hardware as a musician played them on a keyboard. Naturally, this would provide the best possible feedback to the musician. Unfortunately, for some unknown reason(s), note on/off pairs were lost, causing chords to sound as one note. Additionally, note releases were sometimes missed completely. Before long, other unwelcome behaviors surfaced. We worked around these mysteries by having the musicians capture the sample set and play it solely on their keyboards.

After some experimentation, though, we felt that the MIDI music was good, but didn't capture the essence of the John Williams orchestral soundtrack that is so closely associated with Star Wars. Furthermore, each additional instrument channel would require more CPU time than we wanted to allocate.

At this point, we tried an experiment using uncompressed digital samples of the Star Wars main theme. The quality was extremely good, even after subsequent compression with the ADPCM encoder provided by Nintendo. After a little persuasion, Nintendo generously agreed to increase the amount of cartridge space from 8MB to 12MB. This allowed us to include approximately 15 minutes of 16-bit, 11khz, mono music that sounded surprisingly good. Considering that most users would listen to the music through their televisions (rather than a sophisticated audio system), the results were close to that of an audio CD, thereby justifying the extra cartridge space required.

Art Path

A continuing problem throughout the development of SHADOWS was the inability to import and export data between the various 3D packages we were using. Eventually, we managed to circumvent these problems with a number of translation utilities as well as by using Alias Power Animator as our central "hub" format. However, there were still issues with scale, model

hierarchies, and animation data. It was sometimes difficult for the artists to see what their artwork really looked like until it had been through the hands of our polygon wrangler (thanks Tom!). Initially, it was difficult for our texture artist to visualize the restrictions on texture size required by the hardware, as well as color reduction issues.

New Hardware

There were a number of other issues that we had to deal with in developing the game, not the least of which was that for the first nine months of the project, we didn't have any real hardware on which to run the game. This deficiency wasn't insurmountable by any means, but it restricted our choices in certain ways, especially in level design. We were forced to make some assumptions, especially regarding to performance. Fortunately, this wasn't quite the bugbear that we anticipated. Still, as is well known, those on the bleeding edge of technology are often sacrificed upon it.

Other Issues

There was considerable pressure to finish the game in time for the Christmas 1996 deadline. This reality meant many, many late nights, with some team members regularly working over 100 hours every week for the best part of a year. Hopefully, this sort of workload can be avoided in future projects. Time pressure is, of course, a common thing in the computer games industry — and we were certainly no strangers to the phenomenon. However, since we had to release our game shortly after launch of the machine, we were under more pressure than might usually have been encountered. Game testing also became an issue because there were very few machines with which to actually test the game.

Game Play Variety

We were able to include a very wide variety of game play styles in SHADOWS. In retrospect, this meant that we couldn't tune each type of game play as much as we would have liked. It also meant an almost Herculean programming task in trying to write and debug what amounted to five different game engines. These consisted of low flight over terrain, gunnery action in space, first/third person on foot or with jet pack (including a moving train sequence), high-speed chases on a speeder bike, and full 360-degree space flight. Nonetheless, the result was that most players' experiences with the game were always interesting, at the expense of displeasing some of the more hard-core game players. A variety of game play was important for a game that, for many players, would be one of their first experiences in a fully 3D environment.

Hardware Performance

As mentioned before, for the first nine months of SHADOWS, we had no real hardware with which to gauge the performance of the game — other than a rather nice Silicon Graphics

The Core Team

The core team developing SHADOWS from inception to completion consisted of mainly six people, although twenty people contributed to the game for varying lengths of time, and to varying degrees. Nonetheless, everyone played a vital role in the production of the game.

Game Designer/Lead Artist - Jon Knoles
Project Leader/Senior Programmer - Mark Haigh-Hutchinson
Technical Lead - Eric Johnston
Programmer/Lycanthrope - Mark Blattel
Polygon Wrangler - Tom Harper
Level Designer - Jim Current
Level Designers - Matthew Tateishi and Ingar Shu
3D Artists - Paul Zinnes, Andrew Holdun, and Garry M. Gaber
3D Animator - Eric Ingerson
Texture Artist - Chris Hockabout
3D/Background Artist - Bill Stoneham
Storyboard Artist - Paul Topolos
Music Editor - Peter McConnell
Sound Designers - Larry the O and Clint Bajakian
Lead Tester - Darren Johnson.
Production Manager - Brett Tosti

Extra thanks go to Don James, Henry Sterchi, Hiro Yamada, Kensuke Tanabe, and Shigeru Miyamoto. Special thanks as always go to the staff at LucasArts, and particularly to George Lucas for his gift of the Star Wars universe.

Back Row (left to right): Steve Dauterman, Peter McConnell, Jon Knoles, Andrew Holdun, Paul Topolos, Mr. B. Fett; Middle Row (left to right): Jim Current, Matthew Tateishi, Bill Stoneham, Brett Tosti, Ingar Shu, Tom Harper, Chris Hockabout; Front Row (left to right): Garry Gaber, Mark Blattel, Eric Johnston, Mark Haigh-Hutchinson; Not shown: Paul Zinnes, Larry the O, Clint Bajakian, Eric Ingerson, and Darren Johnson.

ONYX. Nonetheless, when we finally received the real hardware, we were pleased to find that the performance estimates given to us by SGI proved to be very accurate. In fact, in large part due to the parallel nature of the graphics hardware, we were able to use floating-point mathematics throughout SHADOWS with no significant impact upon performance.

Additionally, SHADOWS was programmed entirely using the C language — it wasn't necessary for us to use assembler (a first as far as I was concerned, and a pleasant surprise even though I'm a long-time hardcore assembler fan). Since our scene complexity was relatively high (usually kept to around 3,000 polygons or so, but variable according to the level type and design), the graphics task took longer to execute than the program code (that is, we were graphics-bound). Consequently, optimizations to the program code didn't significantly improve overall performance.

NTSC to PAL Conversion

After completing the American and Japanese versions of the game, it was my task to convert the game so that it could run on the European PAL television standard. Being British, I had a vested interest in making sure that the conversion was a good one. This meant two things: first, that the game used the whole of the vertical resolution of the PAL display (625 lines vs. 525 lines of NTSC); second, I wanted to ensure that the speed of the PAL game was the same as the NTSC one, even though the PAL refresh rate is 50hz rather than 60hz.

Fortunately, when we started work on SHADOWS, we realized that one of the most important things to consider was that it had to be a time-based game, rather than a frame-based one. This would allow for update rates that could vary considerably depending upon scene complexity, as well as the simple fact that we didn't have any real hardware from which to measure performance characteristics. Essentially, the program keeps track of the absolute time between each update of the game. This value, which we called delta time, became a multiplicand for any movement or other time-based quantity. By this method, the game runs independent of the video refresh rate, with all objects moving and responding at the correct frequency.

The other issue had to do with the "letterbox" effect that is common to many NTSC to PAL conversions. In most cases, there is no extra rendering or increase in the vertical frame buffer size, leaving unsightly black bands above and below the visible game area. Since the vertical resolution is now greater than the original NTSC display, the aspect ratio will also change, causing the graphics to appear stretched horizontally.

While I wasn't willing to accept this, I had presumed that I couldn't afford the extra CPU time necessary to render a larger frame buffer, even with the extra time available due to the 50hz video refresh rate. There was also a question of the additional RAM usage required by our triple buffering of the frame buffer. My first attempt, therefore, was simply to change both the field of view and aspect ratios of the 3D engine. This simple fix solved the "stretching" problem quite nicely, although the display remained letter-boxed, of course. Unfortunately, it also meant that any 2D-overlay status information remained "stretched." There was the potential that game play could be affected because the field of view, by definition, would affect the player's perception of the 3D world.

Again, this just wasn't good enough. What I needed was a solution that didn't require extra rendering, yet would fix the aspect ratio problems. After a little bit of research, I realized that I had discovered earlier that it was possible to change the size of the final visible display area on the output stage of the display hardware. In reality, it's possible to shrink or enlarge the display both horizontally and vertically. To compensate for the letterboxing, all I had to do was change the vertical display size by a factor of 625/525 or 1.19. Once I did this, I immediately had a full-screen PAL version. Or so I thought….

One of things about SHADOWS is that

we had to compress everything in the game to fit it into the cartridge space available. This included the thin operating system that SGI provides as part of the development system. Therefore, upon machine reset, it's necessary to decompress this OS to run the game. To perform this decompression, we wrote a small bootstrap program, which introduced a small amount of time between the hardware being initialized and the OS starting. This lag introduced a one-time glitch on the screen as the video hardware started. Not very noticeable, except to me. After many late nights, I discovered a way to remove the glitch by directly accessing the Nintendo 64 video hardware registers.

Bad Idea

We then discovered that because we had accessed the hardware directly, it caused an infrequent bug. Rarely (1 out of 50 times) the Nintendo 64 would crash if the reset button were pressed at a particular point in the game. Not only that, I couldn't repeat the bug on my hardware (I hate it when that happens).

After a number of very late nights (over the Christmas holiday), with the help of Nintendo of America's technical staff (thanks Mark and Jim), we finally resolved the problem: first, by removing the code that directly accessed the video registers, and second, by restoring the registers controlling the scaling of the output in the vertical axis upon reset. Sometimes, the simplest solution is the best.

Support from SGI and Nintendo

We were very lucky to receive excellent support from both SGI and Nintendo during the production of the game. The SGI engineers (thanks in

particular to "Acorn") were very helpful and would normally have an answer to our questions within a day, sometimes within the hour. I would like to thank Nintendo for their assistance in the production of the game. Nintendo of America's technical support and QA departments also proved invaluable. In addition, three of Nintendo of Japan's staff spent some time working directly with us at our offices.

I was also fortunate enough to visit Nintendo's head quarters in Kyoto, Japan, to discuss SHADOWS with Shigeru Miyamoto, creator of MARIO 64. His insights were both fascinating and extremely relevant. He is simply a genius with an instinctive understanding of video games.

Of Wampas and Men

When developing a project on the scale of SHADOWS, there will always be some things that didn't progress as smoothly as they could have…

1) The motion capture process proved to be a red herring for us. While originally promising a much more realistic animation solution, in our case the data proved unusable. However, I still believe that it has great potential and deserves further investigation, even though we didn't get to the point of dealing with the potential problems matching the motions to the character's environment and so forth. Caveat emptor.
2) Attempting to use a MIDI-based music solution also proved incorrect for this game. While it promised to be an efficient solution in terms of memory (an important consideration for a cartridge-based game), it simply wasn't suitable for an orchestral soundtrack such as Star Wars.
3) When we started work on SHADOWS, a major problem (that continued throughout the duration of the project) was the inability of various 3D packages to import and export data. Although we were able, for the most part, to write our own conversion utilities, it still proved to be a stumbling block and prevented us from having an efficient art path. Fortunately, the companies supplying these tools now recognize the need for importing and exporting data to other packages, and are taking steps to remedy the situation — VRML, for example, is proving to be a useful format.
4) Time was the biggest enemy of all in producing the game. This is nothing new, but was exacerbated by the fact that we were working on a non-existent machine for nine months. Nonetheless, even though this was, for the most part, out of our control, we were still able to produce a quality game.
5) With hindsight, probably the most important lesson to be learned from the game's development is that of focus. Do one or two things and do them extremely well. Although our ambitions were well placed in trying to provide the player with as much variety as possible, we effectively had to write five different game engines. Additionally, we could have also used a fourth programmer dedicated to all aspects of the front-end of the game; that is, level selection, controller options, and so forth. This would have taken some of the pressure away from the main programmers towards the end of the project.

Out of the Shadows…

Thanks to the talent, dedication, and experience of the SHADOWS team, many things went well during the development process.

1) By using the powerful SGI computers (fairly uncommon in the games industry in 1994) to prototype, combined with our programmers' knowledge of 3D technology, we were able to develop the game rapidly, yet remain flexible in terms of performance requirements.
2) Our ability to reuse tools from our earlier DARK FORCES title saved us time and resources because we didn't have to build all new tools, although a large number of data conversion utilities were necessary. In addition, by reusing familiar tools, our level designers could be more productive earlier in the project than otherwise might have been expected.
3) Our decision to use digitized music proved to be a crucial one. Because most users would listen to the music through their televisions, the quality approximated that of an audio CD as far as many customers were concerned. This alone justified the extra cartridge space required and surprised many players who didn't expect that level of quality from a cartridge game.
4) The conversion of the game for the PAL television standard went extremely well and was much appreciated by customers in those countries. It would be fair to say that SHADOWS has set the standard in that it runs both full screen and full speed. There is no reason why all games from this point on shouldn't run just as well on PAL systems as they do on NTSC.
5) Given that we were working on completely new hardware and for the most part had to discover everything that we needed to know by ourselves, the support from both SGI and Nintendo was invaluable to us throughout the project.

Varying Shadows

Even though we were not able to spend as much time as we would have liked tuning the game, SHADOWS does succeed in supplying the player with a variety of game-play styles. Its popularity is a testament to the creativity and talent of the team of which I was fortunate enough to be a part. ■

by Ron Fosner

Why I Don't Hate Direct3D

It was with some amusement that I read Chris Hecker's "An Open Letter to Microsoft" in the April/May 1997 issue of *Game Developer*. I design 3D graphics software for a living, and frankly, my experiences with DirectX in general and Direct3D Immediate Mode in particular have been less than favorable. When Direct3D made its initial appearance, my first thought was something akin to, "What planet are these guys from?" I've been programming OpenGL on PCs since it first debuted on Windows NT 3.5, so I didn't understand why Microsoft would feel the need to provide another 3D API. I like OpenGL, I wrote a book on programming OpenGL, and I'm happy with it. It's easy to understand, robust, and there's plenty of other folks to ask questions of, plenty of good books on it, and plenty of programming examples. Direct3D has none of that. Microsoft never did provide anything resembling a programming guide, the examples required an Ouija board to understand, and they kept changing the damn interface. All valid reasons to hate it, all acknowledged by Microsoft.

But after I read Chris's article I reflected back upon what a tempest in a teapot it all was. The OpenGL programmers derided Direct3D. Let's face it, the first release of Direct3D was abysmal. The game programmers who started (or were stuck) with Direct3D managed to get some reasonable results in spite of Direct3D's shortcomings, and they, in turn, taunted the OpenGL folks about lack of driver support. Hardware vendors were annoyed because Microsoft was telling them one thing, while John Carmack was proving differently. The 3D graphics industry in general, and game developers in particular, were shouting loudly. Very loudly. All was in a glorious state of turmoil. Now, is this really a bad thing?

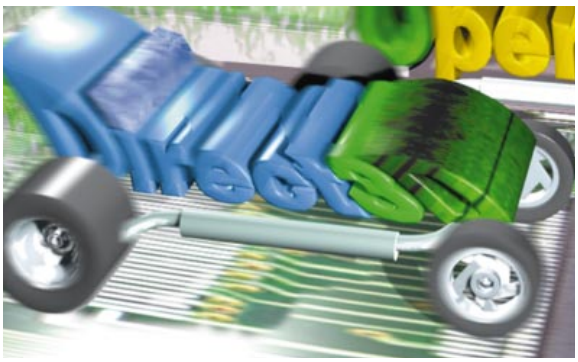
Stop and think for a minute. If you went to the Computer Game Developers' Conference or SIGGRAPH or E3 or Win-HEC last year, you couldn't help but notice all the video card makers demoing both OpenGL and Direct3D software. They were falling over themselves to show you this stuff. They supported Direct3D because they were scared of Microsoft (and they needed to hedge their bets in case Direct3D took off), and they supported OpenGL because GLQUAKE proved that OpenGL was a viable gaming API (and they needed to hedge their bets in case Direct3D flopped). The result of all this is that it suddenly appears that you really will be able to program for either OpenGL or Direct3D, whichever API you like — although because Microsoft has cut IHVs off at the knees by dropping MCD driver support for Windows 95, a lot of the video card makers are scrambling around trying to pick up the pieces.

"Yeah," you might be saying, "there's been a lot of frenetic activity in the past year or so." And you'd probably agree with me that Microsoft instigated the whole thing by bringing out a premature implementation of

Continued on page 63.

Illustration by Rick Eberly.

Contrary to what you may think, Ron isn't a Microsoft employee. He's the author of OpenGL Programming for Windows 95 and Windows NT *published by Addison-Wesley, so his heart is in the right place. Ron is the chief mechanic at Data Visualization, an OpenGL and DirectX custom software/driver house. You can reach him at ron@directx.com.*



Continued from page 64.

Direct3D and touting it as being "The only 3D gaming API you'll ever need." So ask yourself what effect this had on the 3D community. It started the API wars — an endless stream of vitriolic Usenet postings, a flurry of examples and counter examples. It stirred up every video card maker and even a few PC makers. Hell, I don't know about you, but I had a great time!

Since Microsoft made that big boast and then tried for two years to deliver on it, there has been a veritable renaissance of activity in 3D graphics. I used to worry that the adoption of 3D would be slow, that to write a great 3D game would require, unreasonably, that the user have a 3D graphics accelerator. Hah! That's one worry that I can put to rest. This Christmas, practically every PC made came with a 3D accelerator. There were an estimated 40 million 3D accelerator chips sold in 1997. That's a heck of a target market. Just as you no longer have to special order a CD-ROM or a fast Windows video card when you get a PC, 3D chips are replacing the 2D-only systems.

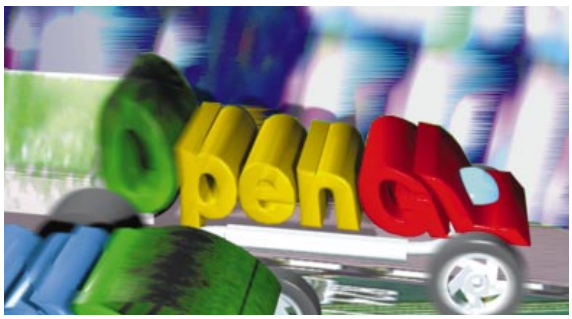
Now, why has this happened? Well, I think it's safe to "blame" Microsoft again. If they hadn't stirred things up as they did, and if the normally sedate 3D graphics community hadn't reacted so strongly, then we might still be looking toward a slow adoption of 3D as a standard feature. Direct3D certainly has lit a fire under the normally sedentary OpenGL ARB. The frenetic pace at which Microsoft is revving the Direct3D API is forcing OpenGL to move at a faster pace. This is good — it used to be that OpenGL was the feature leader. Now, with DirectX 5, we have hardware-optimized texture support (in the form of optimized surfaces). That's a feature that's scheduled for the release of OpenGL 1.2 sometime later this year. And Microsoft has no intention of letting up the pace. They recently gutted their OpenGL graphics group and merged them into the DirectX group (a shotgun wedding, no doubt). What this means is that Direct3D is going to continue to get better, putting more pressure on the OpenGL side of things to increase the pace as well.

So while I agree that last year, Chris raised some excellent points in his article in the OpenGL vs. Direct3D debate, I take the stance that while Microsoft did a bad thing with Direct3D, it did provide a rude wake-up call of the 3D community. Two years ago, most game programmers didn't have a clue what a BSP tree was, what the "A" in RGBA stood for, or how a Z-buffer worked.

Now I see video card makers pondering trilinear and anisotropic texture filters, while Microsoft and Hewlett-Packard are introducing APIs with automatic level-of-detail geometry functionality. These are features that I previously saw only in ultra–high-end systems, that I never would have thought would make it to the PC before the next century. While I still have no love for Direct3D, I admire Microsoft for listening (somewhat) to the criticisms of Direct3D and addressing them.

If the cost of getting 3D accelerators on PCs everywhere by 1999 is that I have to live with Direct3D, then I can live with it. While I personally feel that Microsoft was stupid to bring out Direct3D while they had OpenGL, if they're going to support both of them, then I can live with having two APIs to choose from. I'm not too happy with the active lack of support for OpenGL that Microsoft has demonstrated in the last few months. I think that this is a mistake that will do them more harm than they estimate.

OpenGL on PCs has too much industry support to get killed from lack of attention by Microsoft. Time will tell. Meanwhile, I'm just going to bask in the radiant glow of 3D, giggle when I get a new 3D board to test, play some 3D games and admire those nice multipass rendering tricks, write some code that does some effects from Chris Hecker's physics articles, and sleep soundly at night knowing that next year, even better 3D features will be around for me to take advantage of. If I have to blame Direct3D for causing most of this, then so be it. ■



Editor-at-Large Chris Hecker Responds:

While I agree with Ron's point that the past year has been fun in the roller-coaster sense of the word, I disagree that Microsoft and Direct3D can take credit for the quick adoption of 3D hardware. The truth is, it's simply time for 3D on the PC, and IHVs were already well on their way to making it commonplace by the time Direct3D was even conceived, let alone shipping. I believe a coherent argument can be made that Direct3D's machinations have actually slowed down adoption of 3D hardware, while early standardization on OpenGL would not have had this problem (although I do agree the ARB moves much faster these days). However, hindsight is a wonderful thing. For now — and looking towards the future — I only hope Ron is correct when he says, "You really will be able to program for either OpenGL or Direct3D." I'm not so confident, and I think it will take eternal vigilance from game developers to ensure we have a viable choice of 3D APIs from here on out.